INCA: An Infrastructure for Capture & Access

Supporting the Generation, Preservation and Use of Memories from Everyday Life

A Dissertation
Presented to
The Academic Faculty

by

Khai N. Truong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2005

INCA: An Infrastructure for Capture & Access

Supporting the Generation, Preservation and Use of Memories from Everyday Life

Approved by:

Dr. Gregory D. Abowd, Chair
College of Computing
Georgia Institute of Technology

Dr. Mark J. Guzdial
College of Computing
Georgia Institute of Technology

Dr. James A. Landay
Department of Computer Science & Engineering
University of Washington

Dr. Blair MacIntyre
College of Computing
Georgia Institute of Technology

Dr. William N. Schilit
Intel Research

Date Approved: July 13, 2005

To my parents and my brother,

For all the sacrifices you have made as well as the love and the support you have always provided me.

ACKNOWLEDGEMENTS

An old Vietnamese proverb teaches *"uống nước, nhớ nguồn"* which means *"drink water, remember source."* I could never have completed this dissertation without the help and support of many people. I can only begin to thank them here.

First, I would like to dedicate this work to my parents and my brother, with whom I immigrated to America from Vietnam in 1983. They have always been consistent in their value of education and the pursuit of what will make me happy. I would have never made it through this process without their love and support.

I would like to thank my advisor, Gregory Abowd. Gregory has been a mentor, a friend, and an endless source of guidance and support. I appreciate his enthusiasm, patience, and understanding. I only hope that I can apply what I have learned from him and can make him proud with my own career. I would like to thank the rest of my thesis committee for their support and understanding. Mark Guzdial, James Landay, Blair MacIntyre, and Bill Schilit provided me with invaluable support, advice, and comments on my research and my future. I deeply appreciate the considerable time, interest and effort they each invested to help shape this work.

I also thank Tom Frauenhofer, Patrick Chiu and Maria Pimentel for allowing me to visit their labs and to work with them. I have learned much from each experience; these lessons have forever influenced my approach to doing research.

Although I authored this dissertation, many other people have contributed to different aspects of what is described here. Ahmad Aslami, a dear undergraduate friend,

assisted me in designing and developing early versions of the StuPad application. Bolot Kerimbaev helped me with the creation of several early versions of the abstraction layers ultimately adopted into the INCA toolkit. Molly Stevens, Elaine Huang, and Gillian Hayes have been invaluable to me with brainstorming, designing user studies, and writing and proof-reading countless work.

It is silly to list many of the people that I have been very fortunate to have befriended during graduate school…but I am still going to do it: everyone from the Ubiquitous Computing Group (Anind Dey, Jen Mankoff, Jason Brotherton, Heather Richter, Rob Orr, Lonnie Harvel, Kris Nagel, Rod Peters, Jay Summet, Giovanni Iachello, Xuehai Bian, Gillian Hayes, Shwetak Patel and Julie Kientz), David Nguyen, Joe Tullio, Elaine Huang, David Krum, Jason Elliot, Mike Terry, Kent Lyons, Todd Miller, Carlos Jensen, Duke and Heather Hutchings, Tanisha Hall, and Shanda Harper, you all have meant so much to me. Thank you for being my friends! At times, Anind Dey, Jen Mankoff, Jason Brotherton, and Heather Richter also acted my mentors and I truly appreciate their help and encouragement. I must also express my gratitude to Desney Tan and Jason Hong for their words of advice and support during my last year in graduate school.

Finally, although I have already thanked Gillian Hayes and Elaine Huang, I will do it again. Gillian very quickly became one of my closest friends and I only regret not having met her sooner in life. Elaine is my best friend and I feel very lucky to know her. No amount of words can fully describe how special she is to me. I have been blessed to have ended my stay at Georgia Tech with these two friends whom I truly admire, who greatly inspire me, and whom I will always treasure.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

People's daily lives and experiences often contain memories and information that they may want to recall again at a later time. Human memory, however, has its limitations and many times it alone may not be sufficient. People sometimes have difficulty recalling salient information and can forget important details over time. To complement what they can remember naturally, people must expend much time and manual effort to record desired content from their lives for future retrieval. Unfortunately, manual methods for capturing information are far from ideal.

Over the years, ubiquitous computing researchers have constructed devices and applications to support the automated capture of live experiences and access to those records. At Georgia Tech, we have also investigated the benefits of automated capture and access in over half a dozen projects since 1995. As we encountered challenges in developing these systems, we began to understand how the difficulty of building capture and access systems can prevent exploration of the hard issues intertwined with understanding how capture impacts our everyday lives. These challenges illustrate the need for support structures in building this class of ubiquitous computing systems.

This dissertation presents a set of abstractions for a conceptual framework and a *focused* design process that encourages designers to decompose the design of capture and access applications into a set of concerns that will be easier to develop and to manage. In addition, an implementation of the framework called the INCA Toolkit is discussed, along with a number of capture and access applications that have been built with it. These applications illustrate how the toolkit is used in practice and supports explorations of the capture and access design space.

CHAPTER 1

INTRODUCTION AND MOTIVATION

People's daily lives and experiences often contain memories and information that they may want to recall at a later time. Some examples of specific recollection needs might be:

- While studying for an exam, a student wishes to see all of the class notes that refer to a particular topic.
- On a grocery shopping trip, a customer tries to recall if the family had finished consuming an item earlier in the week to determine if she needs to buy it.
- After a vacation, a person wants to see and to share pictures taken during the trip.

Human memory, however, has its limitations and many times it alone may not be sufficient. People sometimes have difficulty recalling salient information and can forget important details over time.

To complement what they can remember naturally, people must expend much time and manual effort to record desired content from their lives for future retrieval. There are many everyday examples of people manually capturing information for later use. For example, people often take pictures to capture a moment or write notes to record the important information from an experience.

Unfortunately, manual methods for capturing information are far from ideal.

People often have difficulty foreseeing the value of information [Whittaker *et al.* 1994]; *e.g.*, a student might not have written down an important point made by the instructor during class because she may not have realized its importance at the time. Secondly, people also may have difficulty capturing necessary details in a timely fashion. Many times, people simply may not be ready to capture manually the important ideas or events. For example, many parents want a record of their babies' first steps. However, because they can not always anticipate when the event will occur, they may be unprepared to take a picture when the babies actually begin to walk.

People's records often can be biased, incomplete, and in some cases may even contain errors. For example, a picture only captures what a photographer *chooses* to position within a camera's field of view at that particular instant in time. Similarly, when a person takes notes, she often writes down information that *she* finds important. Personal biases in these cases act as capture filters which affect the completeness of the captured records.

The act of manual capture can also distract people from fully engaging in the experience. During the actual activity, people will not be able to pay full attention because they must devote time and effort towards the capture task. Conversely, when people wish to become engaged in the activity, they may not be able to take the time for recording enough details for later use.

The amount of time and effort a person devotes towards the capture of the live experience impacts her ability to recall information during the access phase. When the user does not capture any record, then she will have nothing to review later. However, capture performed over an extended period of time results in a different problem: a

large amount of information that a person must search through later. The user's ability to easily access a specific piece of information depends on not only where the user chooses to store the information, but it also depends on how she chose to organize the content. Retrieval then becomes a matter of how to index into the collection of captured information. However, traditional means for storing captured data often forces information to be filed in one rigid manner. Rigid storage schemes prevent records from being easily cross-indexed. The ability to flexibly index into captured information and to correlate information is important because the salient features for triggering the retrieval of the desired information can be a portion of the content or the surrounding context.

Additionally, access involves more than the simple retrieval of content. Depending on the situation, users may want to recall information at different level of details. For example, when a student reviews a lecture, she may want to listen to what the instructor said on a particular slide. At the same time, rather than listening to everything that was spoken, perhaps a short summarization of a 5-minute audio segment could also be desirable.

1.1 Automated Capture & Access

Increasingly, researchers are applying ubiquitous computing technology to capture details of a live experience automatically and to provide future access to those records. Automated capture can help relieve people from the burdens associated with manually documenting the experience and needing to focus on the recording task. In situations when the act of preserving the details of an experience is not ideal or is

impossible, but records of the experience are desired still, people can simply rely on having important information automatically captured for future access by the computers instrumented in the environment and those carried by the participants. People are free then to fully engage in the activity and to synthesize the experience, without having to worry about tediously exerting effort to preserve specific details for later perusal.

This theme of ubiquitous computing research is commonly referred to as automated capture and access applications. We define *capture* and *access* as the task of preserving a record of some live experience that is then reviewed at some point in the future. Capture occurs when a tool records data that documents an experience. The *captured data* are recorded as *streams of information* that flow through time. The tools that record experiences are the *capture devices*; and the tools used to review captured experiences are the *access devices*. A *capture and access application* can exist in the simplest form through a single capture and access device or in a more complex form as a collection of capture and access devices [Truong *et al.* 2001]. Under our definition, some common tools are already inherent capture and access devices; *e.g.,* pen and paper, cameras and camcorders. However, research in this area focuses specifically on how *automated* capture and access can be achieved unobtrusively by using technology in a natural manner that complements the human activity. Three common characteristics exist across these applications:

1. *Capture should happen naturally.* As Abowd and Mynatt point out, the usefulness of this service lies in its ability to remove the burden of recording from the human so that they can better focus their attention on the activity [Abowd and Mynatt 2000]. As a result, capture must be

supported unobtrusively and can not require any additional effort by the users during an activity. This behavior typically has been supported by 1) capturing raw audio and video of an experience and processing it later for additional semantic information, or 2) augmenting the devices that the user normally uses during an activity to log the user's interaction with those devices.

2. *Information should be accessible with minimal effort.* The design of applications in this area involves more than the development of unobtrusive capture services. Although much of the capture should happen automatically in the user's background during an experience, the usefulness of these services becomes evident in the access phase when users need to review the information. Brotherton previously defined a successful *access* as situations in which information can be found at the proper level of detail (as accepted by the user doing the access) with minimal effort [Brotherton 2001]. Additionally, as Abowd and Mynatt point out, these information access services are most useful when they are ubiquitous [Abowd and Mynatt 2000]. Together, these two points mean that developers must provide users with interfaces that support the appropriate access behaviors when they need it. Designers often have designed ubiquity into these systems by clearly separating the capture and access phases.
3. *Records should be cross-indexed by content and context.* Over a long period of time, automated capture will result in the recording of a large

amount of information. This may cause users to experience difficulty in finding desired points of interest in the captured streams. To help users better navigate through these streams of information, applications often support many forms of indices so a user can jump directly to relevant points in a stream [Minneman *et al.* 1995]. Additionally, access tools should be able to easily correlate and compute over events across streams and levels of detail, as motivated by user retrieval needs. For example, a person might remember a portion of the content or the surrounding context; as such, records should be cross-indexed by content as well as context.

1.2 Exploration Challenges in Automated Capture & Access

Over the years, many researchers have built systems that meet the aforementioned characteristics to demonstrate the benefits of automated capture and access for scenarios such as the classroom, meetings, and other generalized experiences. The creation and evolution of such applications many times includes accidental tasks—activities that developers must address to create a working software system but are not central to the problem domain of the application itself. To overcome the accidental tasks involved in developing and evolving ubiquitous computing applications, developers often must gain sufficient expertise in many different areas of computer science, such as databases, networking, and systems. As a result, *it remains difficult for those with creative ideas for automated capture and access to realize their visions.*

Although application developers can overcome these challenges, the effort to do

so may require much time and effort that they could have devoted otherwise to the design and refinement of the essential features of the application itself. Ideally, adapting the application to support the features required by the user population can be done after the accidental challenges have been addressed. However, developers typically do not leverage existing applications as platforms for further investigation because of the challenges of managing and evolving them. As a result, *existing applications have typically been built as demonstration vehicles, rarely ever evolved into completely functional systems.* Without system longevity, these applications are difficult to maintain and deploy; thus, the actual usefulness of these systems in authentic settings is also hard to evaluate. New and interesting social mores and the synergy developed between the user and the technology are only some of the issues that can be understood and brought about through extended use as both the technology and the users co-evolve [Dourish *et al.* 1996].

1.3 Overview of the INCA (Infrastructure for Capture & Access) Toolkit

We developed the INCA (Infrastructure for Capture and Access) toolkit to facilitate research in this area of ubiquitous computing. INCA encourages a simplified model for designing, implementing and evolving capture and access applications [Truong and Abowd 2004]. The INCA toolkit contains a small set of key architectural abstractions that will help designers rapidly prototype capture and access applications and focus much of the development effort on the essential features of the application, such as:

- *Reusable components to aid in the capture, storage, transduction*[1] *and access of the information.* These four functions are core tasks that often appear in many automated capture and access applications. By providing modules that individually support each of these functions, the primary concerns of a capture application are separated, making the application easier to manage and extend.
- *Attribute-based storage, queries and garbage collection.* INCA treats all captured data as raw bytes tagged with meta-data information. This enables the infrastructure to store and deliver many different kinds of data in the same fashion. Support for the integration of information can be wrapped directly into the access of the information, such that when information is requested, related streams of information are jointly provided. Finally, this provides a way to identify easily the unwanted data to discard.
- *Components to obtain a detailed description of the run-time state of the system and to modify it.* Together, these features allow for implementation of privacy and security features, instrumentation for extended evaluation purposes and dynamic adaptation of application features.

---

[1] We use the term "transduction" to refer to both the act of *transforming* data from one type into another (*e.g.,* converting video into JPEG images) as well as the act of *transcoding* data from one format into another (*e.g.,* converting WAV into MP3).

- *A network abstraction to hide details of the underlying communication scheme from the application concerns.* We implement all the functions described above as specialized modules atop an explicit network abstraction layer. This allows a programmer to develop the appropriate interfaces to address the essential concerns without having to deal with low level networking issues.
- *An extensible library of reusable components that supports the capture and access of common data types, currently including audio, video, ink and Web visits.*

1.4 Purpose of Research and Thesis Statement

In this thesis, we intend to show how the exploration of the automated capture and access design space can be facilitated by providing application developers with the proper architectural support. Our thesis statement is:

> *By identifying and supporting a software structure that separates the core functionalities of a capture and access application and provides additional germane features and abstractions, we can construct an infrastructure that encourages a focused model for designing, implementing and evolving automated capture and access applications. The infrastructure includes a collection of reusable components that addresses the core capture, access, storage and transduction concerns of this class of applications. Additionally, the architecture supports the systematic association of meta-data to content, context-based queries and garbage collection. It also provides the ability to observe and control the run-time of a system. Finally, the infrastructure hides complexities involved with the communication structure and data management aspects of the system.*

Providing developers with novel architectural frameworks, libraries, toolkits and infrastructures to facilitate the construction and evolution of applications is not a new

software engineering practice. When progress in an area of ubiquitous computing slows down, researchers typically then begin to investigate the relevant design abstractions for that well-defined class of applications, develop an architectural solution to support the design and construction of these applications, and then implement a toolkit that embodies these abstractions [Dey 2000; Greenberg and Fitchett 2001; Mankoff 2001]. Many of these projects are then validated through the development of interesting and complex applications within the design space. However, this method for evaluating such research is not widely performed and has not been commonly accepted.

Recently, Klemmer *et al.* proposed a method for evaluating a toolkit by viewing its API as an interface with which developers interact [Klemmer *et al*. 2004]. This method helps in the assessment of the usability of a toolkit in a manner analogous to studying the usability of an application's GUI interface with which a user interacts. Understanding how developers build applications using the API of a toolkit and making it more usable is an important *engineering* component in the development of that toolkit. The *research* value of a toolkit lies in how it addresses the challenges involved in constructing systems in a given design space and how it enables and furthers exploration in that area.

In this dissertation, we present case studies that describe how the INCA toolkit has been used by developers to build capture and access applications. We thereby validated this thesis by demonstrating that:

- *INCA supports the development of capture and access applications.* We demonstrate how INCA facilitates the development of a simple capture application. In a limited deployment, we and other developers have also

used INCA to build many capture and access services. We examined how these applications are composed using INCA.

- *INCA makes complex applications more evolvable and capable of being fine-tuned to meet the users' true needs.* When designers place systems into real use, the user population will often provide feedback pointing to mistakes in the design or additional features that should be added. Two of the systems created using INCA (the Walden Monitor and WebMemex) were extended after they were placed into the hands of the user population. We identified the features of INCA that facilitated the extensions made to each system.

  Because the lifetime of a software system often involves additional developers beyond those who original coded the application, we must also show that features of INCA permit systems to be extended by these new developers. We redeveloped a classroom capture system and asked 15 undergraduate students to extend behaviors in the system. We studied the successes and failures students had with this project. From this evaluation, we gained an understanding of the architectural features provided by INCA that developers found made their task of extending a capture system easier or harder.

1.5 Contributions

We believe this thesis work has resulted in the following set of contributions:

- *The identification of the capture and access application design space.* We

reviewed the literature to gain an understanding of the state of the art for capture and access applications. A design space was identified and used to characterize the kinds of capture and access applications that exist in the literature. Holes in the design space reveal the kinds of interesting applications that have yet to be explored.

- *The development of a conceptual framework and a focused design process that address the minimal set of requirements for constructing and evolving capture and access applications.* From our review of existing work, we also formed an understanding of the common accidental development tasks as well as a set of functional and architectural similarities that exist across these applications. Design abstractions that can lower the barrier for overcoming the difficult accidental development tasks often encountered (handling the tedious and time-consuming low-level construction details) and key separation of concerns based on system similarities between many different applications were used to inform the development of an architectural framework. The programming abstractions and explicit framework allow developers to think about application designs at a higher level—suggesting a simplified and focused design process for developing automated capture and access applications.
- *The implementation of an infrastructure supporting the above framework and design process that acts as a testbed for investigating problems in automated capture and access.* We implemented an infrastructure known as INCA (Infrastructure for Capture and Access) that embodies the

architectural framework and an accompanying design process. This infrastructure provides a number of high-level programming abstractions to "lower the threshold" for rapidly building and evolving automated capture and access applications. INCA also "raises the ceiling" for exploration in this area by enabling researchers to investigate new problems in automated capture and access that were previously unexplored. These investigations include both architectural issues (such as determining what are the necessary and reusable building blocks that would enable application developers to work at a high-level) and application issues (such as exploring holes in the design space with new types of applications that were too difficult to build before).

- *The population of a toolkit with common, reusable capture and access services.* In addition to the high-level programming abstractions supported by the infrastructure, we also provide a set of reusable components to "lower the threshold" for building applications. This toolkit of common capture and access services will help developers avoid duplicating effort or face challenges already previously addressed when building new applications.
- *The construction of a variety of applications investigating portions of the design space previously unexplored.* In the first contribution, we identified the design space for automated capture and access applications. Using INCA, we and other developers have also built a variety of systems that explores different aspects the design dimensions of this application space

(such as number of users, locations, devices, *etc*.). These investigations have furthered the state of exploration in the automated capture and access design space.

1.6 Overview of Dissertation

The remainder of this thesis is organized as follows.

In Chapter 2, we review the related work found in the literature. We extract a set of design dimensions from the list of domain-specific issues that must be addressed for any automated capture and access application. Using these design dimensions, we are able to characterize the kind of work that has been investigated in the past as well as to identify the holes in the design space that remain to be explored.

Chapter 3 presents our experiences building a number of capture and access applications. We discuss the successes and issues involved in the development of these applications. From this discussion, we determine the minimal set of requirements that need to be supported in the building and evolution of capture and access systems and use it to inform the creation of an architectural framework and generalized design process for this class of applications.

Chapter 4 presents INCA, an infrastructure that embodies the architectural framework and implements abstractions that address the development issues described in Chapter 3. We describe the high-level architectural features of this infrastructure, its low-level implementation details, as well as the set of reusable components that are included in the toolkit.

In Chapter 5, we demonstrate how INCA can be used to create a simple capture

and access application, known as the Personal Audio Loop. This application continuously buffers 15 minutes of audio and allows the user to review any portion of that audio. We apply the design process to this problem and illustrate how INCA transforms this design into executable form.

In Chapter 6, we present our various validation activities. This includes descriptions of the numerous applications that we and others have built using INCA. In each case study, we present what the developers built and how INCA was used in the development task. We also describe a structured case study, in which we reimplemented the eClass/Classroom 2000 system and studied the ability of others to build applications on top of a system previously developed using INCA.

In Chapter 7, we summarize the success and challenges encountered by the developers and discuss how to iterate upon the design of INCA based on the observed success and failures of these projects.

Chapter 8 contains a summary and our conclusion. We discuss opportunities that remain for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we provide a historical review of the related work in the area of automated capture and access. We begin with a review of early research visions that have motivated many subsequent explorations in this area. We then discuss projects that researchers have developed and studied. Finally, we present the commercial products that have been marketed, supporting these concepts.

2.1 Inspiring Research Visions

Vannevar Bush was perhaps the first to write about the benefits of a generalized capture and access system. In his 1945 *Atlantic Monthly* article, he described his vision of the *memex*—a system intended to store all the artifacts that a person comes in contact with in her everyday life and the associations that she creates between them [Bush 1945]. In pointing out the need to provide capture and access as a ubiquitous service, he noted that a "record … must be continuously extended, it must be stored, and above all it must be consulted." His envisioned system includes a desk capable of instantly displaying any file and material that the user needs. Bush also envisioned other devices to support automatic gathering of information from other daily experiences for later retrieval, such as a camera that scientists wear on their foreheads to capture pictures during an experiment and a machine to record audio dictations. The goal of these envisioned devices is to support the automated capture of common everyday experiences for later review, a concept reflected in much research since his article.

In 1960, J.C.R. Licklider presented his vision of the "mechanically extended man," describing how man-machine symbiosis can augment the human intellect in the future by freeing it from mundane tasks [Licklider 1960]. He emphasized the importance of the separable functions between the user and the computer in the symbiotic association based on what humans are good at (*e.g.,* synthesizing an experience, making associations between information) and what computers are good at (*e.g.,* capture, storage, *etc.*). Although he and Bush shared very similar visions, the technological progression over the 15 years in between their writings helped Licklider to ground his idea with an understanding of the relevant issues and challenges that needed to be investigated, such as how to store the information and how to provide natural, ubiquitous interfaces for input and output tasks.

Like Bush and Licklider, Douglas Engelbart also believed that technology can be used to augment human intellect [Engelbart 1962]. However, Engelbart believed in more than merely augmenting individual memory. In his later work at the Bootstrap Institute (http://www.bootstrap.org/), he coined the term "Collective IQ" to describe how a group can "leverage its collective memory, perception, planning, reasoning, foresight, and experience into applicable knowledge" to solve problems. The key factor in a Collective IQ is the quality of the group's knowledge repository. However, augmentation also depends heavily on the speed and ease of creating and retrieving knowledge. Engelbart demonstrated the importance of these factors through the simple example of how tying a brick to a pencil can de-augment users.

These early visions were (and remain) inspirational to many areas of computer science. However, it was not until the beginning of the 1990s that these ideas were

explored away from the desktop computer. In his seminal 1991 *Scientific American* article, Mark Weiser describes a vision of *ubiquitous computing* in which technology is seamlessly integrated into the environment and provides useful services to humans in their everyday activities [Weiser 1991]. Weiser described several scenarios demonstrating the benefits of automated capture and access, such as:

1. *Sal doesn't remember Mary, but she does vaguely remember the meeting. She quickly starts a search for meetings in the past two weeks with more than 6 people not previously in meetings with her, and finds the one.*
2. *Sal looks out her windows at her neighborhood. Sunlight and a fence are visible through one, and through others she sees electronic trails that have been kept for her of neighbors coming and going during the early morning.*

These scenarios illustrate two interesting ways that information automatically captured on behalf of the user could be used later for two different environments—at home and at work. However, Weiser left out many important details to inspire others to investigate creative applications for automated capture and access. The first scenario describes the user searching through a list of captured meetings for a particular meeting that satisfies the salient context about it that she remembers. However, we can imagine other desirable access behaviors that could have also helped Sal, such as content-based retrieval or browsable summaries of meetings. The second scenario, in addition to capturing non-traditional data types, demonstrates a very short-term access of the captured information, where walk trails are displayed hours afterward. In this application, the captured information is used only a short time after it occurred; it is

conceivable this captured information is useful even after a long period of time passes. This scenario also introduces interesting privacy concerns relevant to the area of automated capture and access.

2.2 *Early Investigations of Capture & Access in the Workplace from Xerox PARC & RXRC*

Although Weiser's descriptions of these applications were vague, there were real applications behind the concepts that he described. Weiser's article actually previewed many ubiquitous computing projects conducted at Xerox PARC under his supervision as well as those at Xerox's research facility in Europe (Rank Xerox Research Center, RXRC). Because Xerox has a vested interest in technology for the workplace, these projects focused on capture and access for meeting rooms and office environments.

In the first scenario above, the ability to recall who a person has met in the past was based on an application developed by Mik Lamming and Mike Flynn. Their application, Forget-Me-Not, was the first to demonstrate the continuous capture of information for a user as she moves about an instrumented capture environment, the office [Lamming and Flynn 1994]. Designed to leverage the ParcTab devices that a user had with her constantly [Schilit *et al.* 1994], Forget-Me-Not continuously stored the user's location and the people she encountered. Additionally, the application captured the user's workstation activities, file exchange and printing activities, and phone call activities. Forget-Me-Not included an interface that allowed the user to navigate through the captured history or search for documents and other captured data. Users could also apply context filters to narrow their search.

Researchers at RXRC also developed many other systems to enhance a user's retrospective memory and prospective memory. Newman *et al.* developed the Pepys diary application [Newman *et al.* 1991] to collect people's movements around the office. Using the Olivetti infrared active badge network [Want *et al.* 1992], the application simply logged raw location data. Techniques were then developed to extract significant episodes from the raw data. Because episodes recognized by Pepys were purely location based information, they lacked details that might be considered useful. Included in the application was the ability to augment the Pepys Diary with video snapshots [Eldridge *et al.* 1992]. This Video Diary relied on a video network [Buxton and Moran 1990] instrumented throughout the office and controlled by the active badge network to take a snapshot of a person using a camera closest to her location when a significant episode is recognized. Other retrospective memory supporting applications included NoTime, a note-taking application that captures and synchronizes the user's handwritten notes with audio and video of a meeting [Lamming 1991], and Marcel, a system that monitored paperwork activities at an actual desk using overhead video cameras and computer vision [Newman and Wellner 1992]. This collection of research projects, in addition to influencing the design of the Forget-Me-Not application, also resulted in a number of general guidelines for building memory prostheses. These guidelines include the need for sensing to exist in the physical environment, support for both automatic as well as manual data capture, and the development of access interfaces that facilitate finding relevant information. Lamming *et al.* emphasized that "a successful memory prosthesis will integrate seamlessly into the user's normal everyday activities and be available to provide help when needed" [Lamming *et al.* 1994].

At Xerox PARC during this time, researchers also began to investigate the benefits of capture and access for small working meetings. Such meetings are held typically to coordinate team projects and support communication between the project members. As a result, meetings are rich in knowledge that has been transferred between the participants. Manual capture, in the form of note-taking, is one way to preserve content that may need to be reviewed at a later time. However, the task of taking notes can interfere with the participants' opportunity to fully engage in the meetings. As a result, researchers began to develop a variety of applications aimed at exploring the benefit of automated capture and access for meetings.

Perhaps the most influential meeting capture system thus far has been the Tivoli application [Pedersen *et al.* 1993]. Built specifically to capture small co-located group meetings, Tivoli focuses on the capture of interactions centered on large whiteboards. The Tivoli application runs on an electronic whiteboard, known as the LiveBoard [Elrod *et al.* 1992]. In the degenerate case, Tivoli captures the user's annotations as raw ink. The user can also interact with the ink annotations as "domain objects," which represent specific kinds of captured information (such as Intellectual Property). Additionally, the Tivoli application interprets special strokes or gestures and allows users to manipulate the interface; in this case, to create, edit and otherwise interact with the domain objects [Moran *et al*. 1998].

Beyond the whiteboard application, Tivoli has been extended in a number of ways. These extensions include audio salvaging capabilities, supporting ways to create annotations and indices for better playback of the captured content [Moran *et al.* 1997]. In this particular work, Moran *et al.* provided some initial understanding of how a user

would further organize and structure the information. Their evaluation was the first to study how capture and access is actually used in an authentic setting. It is also one of a small number of studies that has been performed to date. This work also demonstrates advanced access services such as information summarization.

While Moran *et al.* explored the benefit of long-term capture and access with the Tivoli system, researchers at Xerox PARC also explored the potential benefits of near-term reminder systems. Minneman and Harrison developed the Where-Were-We application as a service that captures video of the meeting activity [Minneman and Harrison 1993]. Although the content can be reviewed at a much later time, this service also allows users to index into the captured streams when they need to be reminded of certain pieces of information *during* the activity. Later, Tivoli was combined with the Where-Were-We application to form Coral, an infrastructure that explores coordinating a defined confederation of tools to capture collaborative activities for later access [Minneman *et al.* 1995]. This integration was performed using the Inter-Language Unification (ILU) project [Janssen 1994], which provides a distributed-object programming facility. This mechanism allowed developers to create objects that essentially acted as proxies, connecting heterogeneous components together in the larger system. In Chapter 3, we discuss why this approach is sufficient for supporting small changes to a system, but can become problematic in the long run.

Concepts demonstrated in many of the above projects are also seen in other projects investigated in places beyond Xerox. The three similar themes of exploration that have emerged are:

1. The buffering of audio and video for quick, near-term review, similar to

the concept seen in the Where-Were-We application.

2. The capture and access of whiteboard activities inside meeting rooms, classrooms, and offices, similar to the concept seen in the Tivoli application.
3. The continuous tracking and recording of a mobile user's episodic memory in instrumented capture environments, similar to the concept seen in the Forget-Me-Not application.

2.3 Buffering Audio & Video for Quick, Near-Term Review

Prior to Xerox PARC's development of the Where-Were-We system, Hindus and Schmandt created a near-term audio reminder service, known as Xcapture [Hindus and Schmandt 1992]. This service provided a "digital tape loop" of audio for a single office. It was later augmented to provide a short-term audio memory of telephone conversations (5 to 15 minutes long). This application runs on a workstation that captures the phone line and allows the user to quickly review audio content as well as to mark important snippets in the audio loop to save permanently.

Although the Xcapture application buffers phone conversations, the user must interact with the captured audio at a workstation, away from the phone. The intended interaction seemingly is for the user to review important portions of the audio after the conversation ends. Dietz *et al.* later provided a similar capability using the phone device itself as the interface for interacting with the captured audio [Dietz and Yerazunis 2001]. MERL's real-time audio buffering technique does rely on a computer on the back end to record the phone conversation. However, Dietz *et al.* modified the earpiece

to a phone itself to include a capacitive proximity sensor to determine when the phone is near the ear. A change in capacitance indicates the user's desire to relisten to the previous few seconds of the discussion. The user can repeatedly tap the phone to move successively further back in time. This application continues to record the other participant in the phone conversation while the user is reviewing the audio. The application allows the user to catch up to the live conversation by speeding up the playback using audio processing techniques.

As the mobile phone continues to increase in processing power, the phone itself can be used to record audio conversations in the place of a workstation on the back end. Using the Motorola i730 phone, Hayes *et al.* explored the potential usability, usefulness and acceptability issues involved with the user always being able to review audio from her recent past [Hayes *et al.* 2004]. The Personal Audio Loop application continuously records audio from the phone's microphone and stores the audio for a user defined amount of time. When the user presses any button on the side of the phone, the application queues the recorded audio from the previous 30 seconds and begins playback through the phone's speaker. The user can continue to press that button to jump further backwards. The user can press the other button on the side of the phone to jump forward by 7 seconds.

Video content also can be buffered, as demonstrated by the Where-Were-We application. However, instead of acting as a short-term memory aid by preserving recent content that users can quickly review, the StartleCam application buffers video to capture interesting images of a user's surrounding after the system has detected a change in the user's emotional state [Healey and Picard 1998]. The user wears a

galvanic skin response reader that continuously monitors the user's affective mood; when the application detects a deviation, it triggers a camera worn by the user to capture images that caused the response. To compensate for latency caused by the sensor as well as the processing of the data points, StartleCam buffers a very short amount of video content to allow the application to grab images from seconds ago, when the trigger point occurred.

Instead of buffering continuous video, the What-Was-I-Cooking application (also known as Cook's Collage) explores the use of collage displays to show recent activities in the kitchen [Tran and Mynatt 2003, Tran *et al.* 2005]. Through wizard-of-oz techniques, members of the evaluation team select key snapshots of what a subject does in a kitchen during an experiment (that asks the subject to cook). The evaluation team member also annotates the snapshots with useful information (such as the ingredients or the utensils being used by the subject). The application then adds the key frame to the collage which only displays the six most recent snapshots at a time. In the case of a memory lapse or interruption, the subject can rely on the collage to remind her of the last few steps that has been performed. Tran and Mynatt envisioned this application would be useful to a parent with young children in a busy household.

The near-term capture and access applications discussed above capture either audio or video of an activity and allow the user to index into the captured streams when she needs to be reminded of certain pieces of information during an activity. When the access of information occurs shortly after it has been captured, the capture application does not need to support persistent storage. Often, a near-term capture application discards information older than a specified period of time. However, the Cook's Collage

demonstrates a second method for automatically discarding information—by treating the buffer as a fixed size data queue. If the application captures new information and the buffer has reached its capacity, older information automatically gets discarded to make room for the new content.

2.4 Capturing Whiteboard Activities inside Meeting Rooms, Classrooms, and Offices for Later Access

Researchers also commonly explore the capture and access of discussions centered around whiteboards and presentation surfaces inside of meeting rooms, classrooms and offices. The first such application, Tivoli from Xerox PARC, supports the capture of small co-located group meetings. As we described previously, the system provided users with the ability to review pages of recorded whiteboard activity, typed notes from the meeting, and the recorded audio. Additionally, it includes audio salvaging capabilities that allow the user to create annotations and indices. This feature improved the playback of the captured content. Furthermore, it enabled the users to organize and structure the information, as well as to create their own interpretations and summarizations.

The Tivoli application has two major contributions. In its first contribution, the Tivoli application introduces the concept of "domain objects"—representations of very specific kinds of captured artifacts. As a whiteboard surface, Tivoli supports the capture of ink annotations. It also recognizes special strokes and gestures. In the simplest case, this feature allows the user to create, edit, manipulate and relate the contents on the interface. With an additional understanding of the application domain, Tivoli would be able to operate on the ink annotations; for example, the application could recognize

numbers and automatically add them in a math problem.

Secondly, Tivoli is one of the few systems that have been deployed and studied over a prolonged period of use, and the evaluation results have been reported. Moran *et al.* observed the behavior of a single user reviewing information to write reports of intellectual property meetings [Moran *et al.* 1997]. Their study uncovered several salvaging strategies employed by the user to prepare reports about the meetings; and furthermore, it provided an understanding of how these behaviors changed over time. Moran *et al.* uncovered that the user eventually adopted salvaging behaviors on top of the features supported by the interface. The user created marks during the meeting to help him later index into exact portions of the audio that he would review. This study demonstrates the importance of evaluation because, in this instance, the user adopted an unexpected use of features provided by the interface.

In addition to capture and access of discussions centered around whiteboards and presentation surfaces becoming a common theme of exploration, the two contributions above have also inspired a number of other projects. We now describe projects that explore preserving details of the user's experiences in meeting rooms, classrooms and offices for later access.

2.4.1 Classroom and Lecture Environments

As technology has been introduced into classrooms, instructors are given the ability to present more information during each lecture, with the goal of providing a richer learning experience. As a result, students are often drowned with information and forced into a "heads down" approach to learning. While students are busy copying

down everything presented in class, they are potentially distracted from paying attention to the lecture itself. An instructor produces a lot of artifacts while teaching (lecture slides, handwritten annotations, and spoken words), which students attempt to preserve in their notes.

The eClass project (formerly known as Classroom 2000) aimed to alleviate some of the student's burden by recording much of the public lecture experience [Abowd 1999; Brotherton and Abowd 2004]. To capture what the instructor writes, electronic whiteboards, such as the LiveBoard [Elrod *et al.* 1992] or the SmartBoard (http://smarttech.com) are employed. Prepared presentations can be automatically converted into slides displayed on an electronic whiteboard that can be written on. To capture what the instructor says and does, the classroom contains microphones used to record the audio and a single camera to capture a fixed view of the classroom. Finally, to capture other web accessible media the instructor may want to present, a web proxy was used to monitor and record the web pages visited during each class. Immediately after each class, all the different captured streams of information are processed to create an on-line multimedia-augmented set of lecture notes in a form that supports student review [Brotherton *et al.* 1998]. Storing the notes on the Web also allowed students to review the notes at their own convenience.

The work above demonstrates how capture and access applications typically comprise a confederation of components that must work together seamlessly [Minneman *et al.* 1995]. As a result, application developers spend a lot of time creating the glue that allows these independent, distributed, and heterogeneous systems to work together.

Other work that investigates the recording and playback of presentations include STREAMS [Cruz and Hill 1994] and Authoring on the Fly [Bacher *et al.* 1997]. The STREAMS work introduced a technique for capturing multiple streams of information as separate, single medium streams that can be temporally integrated. As many streams of information as possible can be captured for later access, when the user may decide which stream to focus on as the most significant. The Authoring on the Fly system also captures any programs running on a computer as a multimedia document for later playback.

Rather than instrumenting the classroom with augmented capture devices (such as the LiveBoard as an augmented whiteboard and PCs that pull web pages from a logging web proxy), the Lecture Browser application [Mukhopadhyay and Smith 1999], AutoAuditorium [Bianchi 1998], MSR's automatic camera management system [Liu *et al.* 2001], Virtual Videography [Gleicher *et al.* 2002], and other whiteboard applications such as the ZombieBoard [Black *et al.* 1998] and BrightBoard [Stafford-Fraser and Robinson 1996] rely on cameras and vision techniques to capture the materials written and presented on the boards, as well as to detect changes. Additional cameras can track people and provide different images or videos of the classroom that are integrated with the captured presentation when the system automatically generates a multimedia document for the captured lecture experience. Beyond just snapshots of the whiteboard, the Lecture Browser provides a structured interface for accessing the captured lecture; the interface uses a timeline that facilitates temporal navigation through the lecture as well as non-sequential indexing into the content. Work performed at Microsoft has investigated the summarization of the captured presentations as well [Liu *et al.* 2001].

The tradeoff between these two approaches (the instrumentation of the classroom with augmented capture devices versus passively capturing using cameras and vision techniques) lies in the granularity of capture as well as the level of intelligence built into the capture systems. Systems using augmented physical objects the user interacts with which are able to obtain a finer level of granularity in the interaction history without needing to apply much intelligence into the system. For example, when the instructor writes on an electronic whiteboard, stroke level information can be easily obtained. Capture devices that rely on machine vision face a greater challenge to extract this level of information. For example, occlusion by the lecturer can prevent the system from seeing all of the writing as it is being written. As a result, the change detected is not at the stroke level, but at a cluster level (or a coarser level of granularity). Virtual Videography overcomes occlusions in one camera view by using multiple cameras strategically aimed at the same location. This solution assumes that at least one camera will have an unobstructed view of the scene; this situation may not always be true and furthermore the solution does not scale well. The Virtual Videography technique also post-processes captured video after the live experience. Because this processing occurs after the experience, the technique can rely on footage of the scene after the instructor moves away from the whiteboard to compute the ink written by the instructor and uses this information to fill in gaps in the captured video where she occluded the whiteboard.

To support the personalization of the captured lecture experience, the Student Notepad (or StuPad) system [Truong *et al.* 1999] was added to the eClass system to provide students with an interface that is capable of integrating the prepared

presentation, digital ink annotations and Web pages browsed from the public classroom notes into each student's private notebook (during the capture phase). Students' desks were instrumented with networked video tablet technology supporting the act of writing (which is more natural for students to perform and less distracting than typing). Outside of class, it is hard to predict when and where students will review the notes; therefore, the access application was designed to run on networked computers with the more traditional keyboard/mouse interface. The personalized notes are reviewed over the Web to facilitate students to be able to review the notes anywhere anytime.

The DEBBIE system [Berque *et al.* 1999] employs much of the same features supported in StuPad. In this system, what an instructor presents on the electronic whiteboard is broadcasted to computers at students' desks and is added into their electronic notebooks. Both systems have separate areas for private and public annotation, but StuPad allows students to add their annotations on top of the instructor's slides. The DEBBIE system was built as one large application that starts with the capture of the instructor's lecture as well as the students' personal annotations, while StuPad extended the base eClass system—exploring the personalization of public experiences.

The Audio Notebook is a private device used to capture an experience for just its user [Stifelman 1997]. This electronic notebook supports the recording of audio, which it integrates automatically with handwritten annotations. The Audio Notebook also includes technologies such as page detection and an annotatable timeline to facilitate the access of specific portions of the captured experience. Because all information captured by the audio notebook effectively reside on the same device, information streams do not

need to be integrated. However, the device uses time to synchronize and index into the captured information for playback. Because it is a mobile device, the Audio Notebook could support the capture of almost any general user experience. In her dissertation, Stifelman demonstrates the general purpose ability of this device through the study of its use in three domains: when a student uses it to take notes during class, when a journalist uses it to capture interviews, and when she uses it to take mundane everyday notes [Stifelman 1997]. She later analyzed the results of this study and provided an understanding of how users, two students and two reporters, coped with the interface to retrieve the information they wanted to review [Stifelman *et al.* 2001]. This 5-month long longitudinal study resulted in several key findings that can be applied in new note-taking devices. First, users often skim through notes and only review in full detail the material that was unclear during lectures or meetings that was unclear. This discovery suggests the need for the ability to skim audio at faster than normal speed. Users also noted that playback often started in the middle of a phrase when they index the audio using page level or ink indices. This finding resulted in the development of an "audio snap-to-grid" feature using phrase detection. Stifelman *et al.* also processed the audio to predict topic introductions through an analysis of the pitch, pausing and energy. The Audio Notebook device indicates changes in topics on a physical scrollbar that allows the user to snap quickly to new discussion threads within the recorded meeting or lecture. Not surprisingly, Richter *et al.* found similar user behaviors and needs in their TeamSpace project, which we will describe in the next section [Richter *et al.* 2001]. In contrast to the Audio Notebook, the TeamSpace project ran for a longer period of time and supported more captured sessions and users.

2.4.2 Meeting Rooms and Offices

Meetings are an important communication and coordination activity that occurs for a group of people. During such an activity, much knowledge is transferred between the participants. As a result, meetings are rich in content that might need to be reviewed in the future. Because discourse and communication are important during meetings, often the task of taking notes can interfere with the participants' opportunity to fully engage in the meetings themselves. As a result, a variety of applications have been built over the years to explore the capture and access of a number of different types of meetings.

The SAAMPad application is a system that supports the capture of software architecture discussion sessions [Richter *et al.* 1999]. During these sessions, the rationale behind the architecture of a system is presented verbally and through architectural diagrams. The diagrams and discussions are, therefore, important aspects of the meetings that are captured and related later on. An electronic whiteboard is instrumented to capture the information that would normally be presented on a public display and ties that information with additional captured audio and video streams. Because it is known ahead of time the kind of annotations users would draw on the board, the application supports application specific objects, such as types of architectural blocks in an architectural diagram.

Similarly, in the domain of military strategic planning sessions, users would navigate over a high definition map to place annotations of military symbols on it as they present a strategic operation. This information is best captured in a way that supports full playback to allow those not present to hear how the operation should play

out and the rationale behind some of the tactics. Because the set of annotations drawn over the map belong to a well-defined set of symbols, it is possible to interpret these ink strokes into recognized objects. The Rasa system [McGee *et al.* 2000] applies vision techniques to capture and interpret a strategic planning session discussed over both SmartBoards as well as actual paper maps.

Despite their support for special domain objects, SAAMPad and Rasa capture information in similar manners to eClass and the Lecture Browser, respectively. In these applications, again we see the practice of either augmenting physical devices or using vision techniques previously discussed to capture information being applied to a different domain.

Project group meetings are used to discuss various aspects of a group project. Meetings can be devoted to understanding the team's progress; specifics on how important parts of the projects are implemented (or will be implemented) are sometimes presented, agendas are drawn out, and schedules and responsibilities are defined. The TeamSpace project [Richter *et al.* 2001] supports the capture of these meetings as multimedia meeting notes as part of a larger set of shared artifacts created and maintained for each project. Traditionally, these meetings involve multiple people who come together at a mutual location. As companies look to grow world wide, the nature of the work place is now a distributed environment with multiple people at different geographical locations collaborating in a large project. The TeamSpace application can be launched at these different sites involved in the meeting to capture and share streams of information between the remote locations. The different streams of information the application supports include presentation slides, annotations, agenda items, action

items, and video frames. Telephone connections are used to provide an audio connection between these physical spaces. Thus audio is captured through the phone line, although potentially a voice over IP solution could be instrumented as well. Like eClass and Tivoli, TeamSpace is one of the few applications that have been deployed for a prolonged period of time. Richter studied its authentic use and also conducted laboratory studies to understand how users review captured meetings [Richter 2005]. In the laboratory study, she uncovered six browsing techniques (scan, skim, jump, honing, replay, and random). But in her deployment study, she discovered that people rarely reviewed the meetings. Users review meetings when they need to recover important information that could not be otherwise obtained. From her study, Richter concluded the cost of capture must be lowered significantly beyond the need to review. Because review seldom happens capture should done at a minimal cost to the users, but capture must be performed so that it is beneficial the few times the users do review information. Richter also created the tandem Tagger and TagViewer applications to study the capture of meetings where users have a high need for such records. The Tagger application allows software engineers to annotate a text transcript of a recorded requirements gathering session. The TagViewer application allows users to review the requirements gathering sessions. In a controlled study, Richter studied the effect of the TagViewer application versus just the captured video that software engineers typically use today. She reports that users extract nearly the same number of requirements from both, though there are fewer errors when using the TagViewer application.

There are a number of other applications that provide similar functionality to TeamSpace. For example, the Workspace Navigator application from Stanford [Ju *et al.*

2004] captures student project group meetings within a dedicated physical workspace instrumented with cameras that capture snapshots of the room, whiteboard, and physical objects. The Workspace Navigator also recorded screenshots from the computers. The system helps students record design information for later reuse.

The Magic Lounge application [Rist *et al*. 2000] also investigates the problem of preserving a memory of the activities that go on during meetings that exist at virtual meeting places. Different from TeamSpace, this work is less concerned about the generation of multimedia presentations. Instead this work focuses more on how issues such as recording and storage, grouping, indexing and linking, extraction and summarization, and interpretation and reasoning must be addressed to cater to three useful kinds of memory components of the users: short-term working memory, private memory, and shared memory.

In comparison to other meeting capture and access applications, TeamSpace supports the capture a single collocated collaborative meeting but also has support for multiple people to collaborate in the capture of the streams of information. Similarly, the DOLPHIN application [Streitz *et al.* 1994] was designed for capturing small free-form meetings where group members may be either in the same room or in different locations taking notes on computers. Like Tivoli, DOLPHIN also supports gesture input for interacting with domain objects created by the user. Different from most capture and access system, DOLPHIN structures the captured information as a hyperlinked set of nodes instead of temporally contiguously sheets or slides of notes. DOLPHIN supports both shared and private annotations in a manner similar to DEBBIE. These systems go beyond just allowing multiple devices to control a single meeting surface

(such as Pebbles [Myers *et al.* 1998]). More compellingly, they provide multiple people and multiple locations with the chance to participate in the capture of information. The key difference is everyone can capture information and it must be shared across all locations.

Personal meeting capture is achieved through personal electronic notebooks that users would have at their seats. The Filochat system [Whittaker *et al.* 1994] and Dynomite [Wilcox *et al.* 1997] are built on pen-tablet computers instrumented with a soundcard. In these systems, notes can be scribbled and synchronized to captured audio content. Dynomite builds on the list of issues uncovered by the Filochat study and as a result has some additional features that include organizing the notes based on user queries and assigning keywords to blocks of ink.

In early phases of website design, ideas of the eventual site structure are collected and arranged through paper Post-It notes on large walls or tables, where large amounts of information can persist for as long as needed. To support this practice in the electronic world (thereby making it easier to share the information as well as to maintain design changes), the Designer's Outpost application uses two digital cameras and a rear-projected electronic whiteboard to bring the artifacts manipulated in the physical world into the electronic world [Klemmer *et al.* 2001]. The whiteboard provides a large augmented surface where notes can be added and links and annotations can be written, much like a wall or a table in the existing practice. When a designer adds or removes paper notes from the whiteboard, the two digital cameras are used to determine the changes and to capture the notes. After each interactive Outpost session, the notes and the links and annotations on the whiteboard (effectively a sitemap) can be

saved into a DENIM input file, while still physically persisting on the whiteboard for as long as needed. DENIM presents the design at many different levels such as the sitemap, storyboard, and page schematic representations of a web site and allows the user to continue to refine the design, making it a convenient (and appropriate) access application of the captured information [Lin *et al.* 2000].

The NotePals application also allows users to each privately capture their notes during the live experience [Davis *et al.* 1999, Landay *et al.* 1999]. After the experience, all the user's notes are gathered to form a collective view of the experience during the access phase. This approach takes into consideration that some points may be missing in different people's notes, or that the users' views may be different. The NoteLook system also supports the integration of both public and private content [Chiu *et al.* 1999]. The NoteLook system provides users with an array of camera views that when a seminar participant requests can be used to take snapshots of the public presentation. Once the snapshot is integrated into the user's private notebook, private annotations can be placed on top of it. The subtle difference between NoteLook and StuPad lies in NoteLook's reliance on the participants to devote effort and awareness (as well as a little anticipation) on when to request the public information to be added into their personal notebook. Additional work at FXPAL has also investigated the automated video capture of meetings and generation of a comic book layout summarization of the session. This application, known as Manga, suggests high-level points of interest in the video through different image processing techniques (such as keyframing) and allows the user to drill down into video to find segments for playback [Uchihashi *et al.* 1999].

Finally, serendipitous encounters present the challenge of not knowing who the

participants are and when the meetings could potentially occur. Because whiteboards are the site of where a lot of these types of informal meetings take place. These boards are often placed in locations where there is a reasonable flow of traffic to encourage anyone who passes to discuss ideas and to brainstorm with one another. The DUMMBO application uses a non-projecting SmartBoard with an attached sound system, to capture informal and opportunistic meetings [Brotherton *et al.* 1999]. When anyone approaches the board and picks up a pen to write, the board automatically begins to capture the writing and discussion. After a certain period of inactivity, recording will stop. Sensing technology is instrumented near the whiteboard to detect the people present during each meeting. If two or more people are known to be near the board, then recording of the conversation will occur even if no writing appears on the electronic whiteboard. A Web interface is provided to support the access of this collection of unstructured meetings. The context of an informal meeting (who was there, when and where it occurred) is used to help an individual find a meeting of interest. Users may browse through a timeline displaying periods of activity at the board and may apply filters (who, where, when) to pinpoint a meeting of interest. Once an appropriate time period has been selected, and the correct meeting has been retrieved, the access interface allows the user to replay the whiteboard activity, synchronized with the audio.

Xerox PARC's Flatland project [Mynatt *et al.* 1999] has also looked at the capture and access of informal activities, and uses time as the mechanism used to retrieve historical information. Flatland was designed to support informal activities within a private office. More structured activities in the office, such as actions

performed on desktop computers, can be logged and visualized in peripheral displays as a montage of images to remind users of past actions. The Kimura project supports this type of capture and access of office activity to assist the user in managing multiple "working contexts" [MacIntyre *et al.* 2001]. Specifically, Kimura allows the user to multitask and to switch between working contexts by moving different tasks throughout the office. Within a task, there may be multiple branches of activities or contexts as well. The Designer's Outpost demonstrates how to manage and visualize a task that branches and has multiple working contexts through a history interface that includes a main timeline, a local timeline and a synopsis view [Klemmer *et al.* 2002].

2.5 Continuous Tracking and Recording of a Mobile User's Episodic Memory in Instrumented Capture Environments

Forget-Me-Not was perhaps the first application to demonstrate the concept of continuously capturing a mobile user's experience in an instrumented capture environment—the office. Many recent applications have revisited the same concept in spaces rich with information, such as museums and academic conferences. User's experiences in these environments can be enhanced when automated capture frees people from the task of needing to manually record of the information they encountered.

Want *et al.* have explored this concept again recently with the Personal Server—a networked device that users can carry with them capable of storing and providing access to all her personal information as needed [Want *et al.* 2002]. This device contains no integral user interface, but instead can wirelessly communicate with input and output devices (via BlueTooth). Want and his researchers are currently investigating an architecture that accumulates wireless signals from devices the user

passes as she walks about an environment.

Academic conferences often have multiple tracks of concurrent activities that include paper presentations, demonstrations, special interest group meetings, *etc*. Conference attendees typically move about and listen to the track they find most interesting. To help remember the presentations they have seen, attendees usually take notes. However, the abundance of potentially novel and interesting information means that attendees may struggle between attempting to take notes or to synthesize the presentations. Furthermore, the large amount of information, ranging over many different topics makes it difficult to organize the notes. Dey *et al.* created the Conference Assistant as a mobile capture and access application that allows users to take notes that automatically integrate with the tracks they attended for later review [Dey *et al.* 1999]. As attendees arrive at a conference, they each receive a handheld PDA for use during the conference. Rather than requiring attendees to take detailed notes, the instrumented environment automatically captures and tags each presentation with the fixed location. Conference attendees can take summary notes on their personal, mobile devices. Because attendees often move about during presentation sessions, the Conference Assistant logs the user's physical location at all times through the use of RF-ID positioning technology. When an attendee reviews a talk she attended, the application uses location information to integrate the personal notes with the actual presentation.

At the Advanced Telecommunications Research Institute International (ATR), Sumi *et al.* explored providing useful information to visitors during exhibition tours of museums, trade shows, conferences, *etc.* as part of the Context-aware Mobile Assistant

Project (C-MAP) [Sumi *et al.* 1998]. Based on the user's profile, which includes age, gender, experience participation type and personal interests, as well as, the user's location, an animated agent character appears on the user's mobile computer to help guide her through the physical space. Later, Sumi *et al.* added the ComicDiary application to automatically generate a comic strip that recounts the conference attendee's experience based on sensed and manually inputted content [Sumi *et al.* 2002]. This application, running on the hand-held device, accumulates personal contextual information, such as the touring history or list of people encountered, and provides the user with an interface for inputting the level of personal preference (for the current situation), current interest, *etc*. A story generation engine creates a story from the captured information and presents it as a comic strip.

Similarly, the HP Remember system allows a museum visitor to author an automatically generated Web page recounting her experience through both sensed and manually added content [Fleck *et al.* 2002]. As she enters the environment, she receives an RFID tag that can be docked at readers placed throughout the exhibit to register her presence. Additionally, the user also carry wirelessly networked Pocket PCs that she can point at and communicate with Cooltown infrared beacons mounted on an exhibit. This explicit action invokes cameras instrumented in the environment to take pictures of the user and the exhibit. A record of the user's museum visit is preserved as a set of web pages that can be reviewed afterwards.

The user's mobility in the situations discussed above requires the capture applications to be also context-aware. These applications use more information than simply time to integrate the appropriate information. In particular, the sensing of people

present in any given location needs to be supported. By automatically sensing or allowing users to input this piece of context, these applications can dynamically integrate all the different streams of information that the user experiences. In contrast, systems such as StuPad and NoteLook support the personalization of captured information in settings such as classrooms or seminars, but they assume fixed user location. The NotePals application has also been used at conferences. NotePals integrates a collection of personal notes based on matches in context, such as location and time. However, it operates independent from any instrumented environments.

2.6 Long-Term Capture & Access

In 2001, Gordon Bell became interested in how the rapidly increasing affordability and size of disk storage can be used to fulfill Vannevar Bush's vision of the Memex. His project, initially named as CyberAll [Bell 2001] and later renamed to MyLifeBits [Gemmell *et al.* 2002], explored issues related to encoding, storing and retrieving all of the user's personal and professional information. Bell provided rough estimates on how much disk space is required to store different media that a person would create or interact with in her life. For example, Bell claimed that 40GB, costing $400 in 2000, would be sufficient for holding all of a professional's lifetime reading, presentations, and audio recordings. Bell also computed the time and cost to capture paper documents, photos, and CDs.

In the early stages of the project, Bell made the decision not to use a database. He believed that the Windows file system organized into a relatively flat three level hierarchy, with about a dozen first level folders and an average of four folders in the

second level, would provided more flexibility than a database. Bell believed that the need to maintain the database columns and metadata was an unnecessary cost. To meet the user's need, Bell believed that applications could support ordinary indexing techniques, such as temporal indexing, and searching by automatically extracting metadata from the documents. Additionally, he feared that continuous personal capture would involve too much variation in document types and he did not think databases could be flexible to the moving or modifying of files.

However, the actual realization of the CyberAll vision as the MyLifeBits project actually used an SQL server that supported full-text search. The schema consisted of a table for the resources, a table for the annotation links and a third table for collection links. The resource table stores information as blobs and has additional columns for standard properties such as type, size, creation date, last modified date and a short description.

To explore storage of information beyond personal and professional content, such as all of a user's reading, presentations, and music, Gemmell *et al.* created SenseCam, a personal, mobile, passive capture device [Gemmell *et al. 2*004]. SenseCam automatically captures images from a person's life without her having to operate the recording equipment. Gemmel *et al.* designed the device to be the size of a pager that could be worn on the front of the user's body via a neck strap. SenseCam includes accelerometers, a light sensor, a temperature sensor, and a passive infrared sensor to detect motion. A PIC microcontroller polls these various sensors every second. SenseCam automatically captures images after a certain amount of time elapses or when triggered by the various sensors. Additionally, the user carries a GPS unit to

record her location information. An import program uploads the images and all sensor data into MyLifeBits. This program automatically relates all properties, including GPS coordinates to the image, instead of storing them as separate streams of information that can be correlated later using timestamp. Gemmel *et al.* made this decision with the assumption that photos can be shared between users. A rapid serial visual presentation (RSVP) access application allows the user to later playback the captured photos and the sensed data. A separate access interface allows users to review information based on location information as well. To demonstrate the performance of MyLifeBits for SenseCam results, Gemmel *et al.* replicated one day's worth of data for a year. This resulted in 318,000 SenseCam samples and over 55,000 SenseCam images. This allowed them to measure processing time for various operations (such as query, sort, *etc.*) performed over the MyLifeBits storage.

Even prior to the development of SenseCam, many hobbyists have carried GPS units when taking pictures in order to tag their captured photos with location information. As GPS units have continued to become smaller and cheaper, manufacturers of digital cameras have begun to package this feature in their products as well (http://www.ricoh.co.jp/dc/caplio/pro_g3/). For cameras on cell phones, location information can be obtained from the GSM network's cell IDs. The MMM prototype uses a simpler form of this information to tag captured photos with crude location information, but also to roughly predict the people's location [Sarvas *et al.* 2004]. By collecting the user location, it is possible to suggest the list of people who may be captured within a photograph at any given location. Because a single GSM network cell ID can only provide crude location information, the predicted list of people present can

be inaccurate.

Researchers, however, have continued to investigate how to capture additional context information that can facilitate in the organization and retrieval of these photos. As previously described, using a galvanic skin response (GSR) sensor, the StartleCam detects changes in the user's emotional state and automatically triggers a camera worn by the user to capture a picture of the surroundings [Healey and Picard 1998]. In addition to detecting the user's emotional state using GSR, users of the LAFCam system also wear a microphone that continuously monitors the audio [Lockerd and Mueller 2002]. An HMM trained to detect laughter processes the audio source and annotates captured video with points in the captured stream when the user laughed. This interface can simplify the video editing process by marking the most interesting parts, as detected by the sensors.

In contrast to the previously mentioned camera projects that capture context as a separate stream of information, Hâkansson *et al.* explored how to add *within* a photo the pieces of context that can be easily sensed, including movements, sound, temperature, pollution, humidity, smell, and electromagnetic fields [Hâkansson *et al.* 2003]. Inspired by lomography, the art of taking photographs that capture spontaneous moments that include unpredictable color and lighting effects using special cameras and a "don't think, just shoot" mentality, Hâkansson *et al.* used sensor data to control the hue, saturation and value of an image, thereby encoding the sensed context into the captured photograph, itself.

Although a photograph or a video clip obviously can capture people within a scene, an access application can not easily extract this piece of context from the content

to facilitate users in searching for the specific picture or video segments they wish to review. Patel and Abowd developed the ContextCam device to capture video that is automatically tagged with context information [Patel and Abowd 2004]. ContextCam detects people present within the field of view of the camera through an active tagging scheme that assumes people wear badges that transmit ultrasound and radio frequency signals. The ContextCam device includes a pair of RF and ultrasound readers along the front of the device to triangulate the distance and position of the signals it reads from the device. Based on the zoom level, the camera can determine the list of people present in its field of view. The camera can also determine the list of people nearby but not in the field of view. ContextCam encodes the list of people it detects as well as time and location information directly into the captured video. The camera modifies every sixtieth video frame. ContextCam encodes metadata information into the least significant bits of the eight bit RGB value for each pixel of the video frame. As a result, how much metadata needs to be stored determines how much the camera adjusts the quality of these video frames. However, users will not notice the modification of the video frame during playback because it is beyond the 29.97 frames per second playback rate typically supported by VHS and DV equipments. Patel and Abowd developed ContextCam as a point of capture device with the intention of creating video content that would eventually feed into the Family Video Archive application. The Family Video Archive application provides the user with an interface for manually tagging captured video with high level context that can not be easily sensed, as well as, an interface for searching the archive [Abowd 2003]. Using a naïve Bayesian classifier, trained on previous captured content and its list of tagged attributes, the ContextCam

also suggests to users a list of relevant metadata information that can be added to the captured video as well.

Although the works described above focus on adding metadata information to captured content to facilitate access, with the exception of the Family Video Archive application, they do not actually investigate the storage and retrieval aspect of the problem. Many of these works simply store the information streams within a hierarchical file structure. However, as Gemmel *et al.* observed, a database back-end provides the most flexibility for storing and organizing captured information. As information retrieval highly depends on what users can remember about a document, property-based storage systems have been proposed as an alternative solution to hierarchical storage systems. Property-based storage systems better support user interaction with documents through document attributes. A number of systems use (or can use) information's natural space and time attributes to manage, organize and visualize documents. Time-based systems, such as LifeStreams [Freeman 1997] and TimeScape [Rekimoto 1999] organize documents based on time and provide special visualizations. LifeStreams archives documents as a time-ordered "stream" of documents that are displayed stacked diagonally across the display. Users can scroll this stack and click on documents to interact with them. TimeScape takes snapshots of the desktop workspace, and the desktop can be "played back." It also has a horizontal timeline view of the documents and uses fading to handle clutter. Both LifeStreams and TimeScape allow users to retrieve retrospective content, but also enable users to create prospective content as well. By inserting information with time anchors from the future, users can insert reminders to act upon at a later time.

In the Presto project, later renamed to Placeless and finally Harland, Dourish *et al.* investigated a more flexible model for interacting with documents that uses arbitrary properties instead of simply temporal context for storing and retrieving information [Dourish *et al.* 1999]. Users can retrieve a document (including those in mail boxes or mounted over network file systems) by specifying a query for known properties of that desired document (*e.g.,* author, topic, *etc*.). The use of properties to store and search captured written notes has been explored in the NotePals application as well [Huang 2000]. However, Presto also supports executable properties which embody autonomous behaviors that help the document space and applications reorganize the documents and the define ways to interact with them.

All the work described above, with the exception of the Family Video Archive work coupled with the ContextCam, simply examine one aspect of the capture and access problem, whether it is: how to capture information, how to store the information or interfaces for accessing and visualizing the captured information. In contrast, Rhodes and Starner developed the Remembrance Agent to examine how all the information a user has previously seen and recorded can be related and made useful to her [Rhodes and Starner 1996]. This application demonstrated the potential benefit of having captured information available after long periods of time beyond the initial live experience. Rhodes and Starner intended for the Remembrance application to run on a wearable computer that the user always has on her body. As the user interacts with a document, the system determines the local context or keywords related to that file and attempts to automatically remind the user of other documents she has viewed in the past that are similar.

2.7 Commercial Capture & Access Products

Many researchers developed projects described above atop a number of commercial products such as LiveBoard [Elrod *et al.* 1992], SmartBoard™ (http://smarttech.com/), and the Mimio™ (http://www.mimio.com). Beyond whiteboard capture surfaces, companies have also produced a variety of capture devices such as the Casio E-Pen™ (http://www.casio.com/index.cfm?fuseaction=products.detail&Product=EPEN-USB), the Anoto® pen (http://www.anoto.com/), the portable HP Capshare copier and the plethora of camera phones. Many of these products were created with the capture service in mind. However, capture software distributed with them differ significantly from the automated capture and access applications and environments we described in this chapter

Recently, many presentation and multimedia applications, such as RealNetwork™ encoders (http://www.realnetworks.com/info/blackboard) and Microsoft PowerPoint, have begun to add support for recording presentations and even ink annotations. Quindi's Meeting Companion™ (http://www.quindi.com/) supports the capture of audio, video, and notes and data, including PowerPoint slides, screenshots and whiteboards. The application records the meeting and creates a single file that is indexed for easy review and convenient sharing by email. In many ways, it provides similar functionalities as the ones offered by the TeamSpace application. The Deja View device (http://mydejaview.com) is a wearable digital camcorder that assumes the user attaches it to the bill of a hat or the frame of her glasses. The device buffers video and allows the user to store captured content from up to 30 seconds in the past. This system enables users to capture video footage of an event after it happens, a concept

similar to what we discussed in Section 2.3. Overall, these products demonstrate that companies believe capture is a marketable service.

Automated capture environments more recently have also become produced commercially. While few of us currently spend time in a fully capture-enabled environment, the trend toward ubiquitous capture is real. A capture-enabled environment contains technologies that record relevant human activities. These include cameras and microphones to record what can be seen or heard, surfaces or devices that store written materials, and devices to access the captured records. The capture and access devices provide full coverage of a particular environment, like a house, a school or an office building. Increasingly, classrooms are becoming capture- enabled due to the positive research results of efforts like eClass [Abowd 1999] and commercial products, such as Silicon Chalk (http://www.silicon-chalk.com), Tegrity (http://www.tegrity.com/) and Media Site, by Sony's Sonic Foundry (http://www.sonicfoundry.com/). Additionally, research efforts like the Aware Home (http://www.awarehome.gatech.edu) and commercial endeavors like Oatfield Estates in Oregon, developed by Elite Care (http://www.elite-care.com), show the promise for capture-enabled environments to facilitate "aging in place."

2.8 Summary

Over the years, many have been inspired by the visions of those such as Vannevar Bush and Mark Weiser. In this chapter, we provided an extensive review of this body of work. From our review of these applications, we believe systems in this application space can be characterized based on the following design dimensions:

1. ***Length of time captured information persists.*** Some applications, such as Xcapture [Hindus and Schmandt 1992] and the Personal Audio Loop [Hayes *et al.* 2004] keep captured content available for only a short amount of time. These applications act as quick short-term reminders for the user. Other applications, such as eClass [Abowd 1999; Brotherton and Abowd 2004] and the Cornell Lecture Browser [Mukhopadhyay and Smith 1999], store information indefinitely, allowing the user to review the content at a much later time after the live experience. These applications allow the user to recall rich details from their past.
2. ***When & where capture occurs.*** Some applications capture experiences that occur within fixed spaces that occur at regularly scheduled times. For example, eClass and NoteLook [Chiu *et al.* 1999] support the capture of lectures and meetings that typically occur in a classroom and meeting room at fixed times respectively. A few applications have begun to explore the capture of a user's experience in a larger space than a single room (such as the Conference Assistant [Dey *et al.* 1999] and the HP Remember system [Fleck *et al.* 2002]) and her experience continuously across multiple environments (such as the Personal Server [Want *et al.* 2002]).
3. ***Number of devices comprised the application.*** Applications that capture experiences within one fixed environment can afford to distribute the responsibility for capturing different streams of information across multiple machines embedded in that space. For these scenarios, the user

typically would use yet another device to review the captured experience. However, applications that follow the user around as a personal service typically run on a single device. For example, the Personal Audio Loop and the Audio Notebook [Stifelman 1997] are mobile self contained devices that support the capture, storage, as well as the later review of previous experiences.

4. ***Methods for capturing and annotating the live experience.*** Capture can be performed passively using cameras and microphones (as done in the Lecture Browser system) or actively by augmenting the tools that the user interacts with inside an environment (as done in the eClass system). Capture, however, is only the first challenge to preserving content in a way that allows users to review information at a later time. Additionally, applications must annotate the captured information in a meaningful manner to facilitate its recall. At the least, applications tag captured content with meta-data in order to distinguish one captured session from another and allow the various streams of information within that captured session to integrate. Many projects have demonstrated how time and/or location are minimally sufficient for this purpose. The Family Video Archive project [Abowd *et al.* 2003] demonstrates a compelling reason for manually annotating captured data with a richer set of meta-data; such annotations can also be performed automatically by the application using sensors and learning techniques [Patel and Abowd 2004].

5. ***Techniques for reviewing captured information.*** A person must use what she can remember about an experience to retrieve specific data from the collection of captured content. The Family Video Archive demonstrates one way an application can support the dynamic searching and grouping of relevant information from a large amount of captured data. In simpler scenarios, the user obtains all the information from a particular captured session for her perusal (meaning the application automatically performs the grouping of captured information at the session level). As shown with the eClass application, one stream of information (ink annotations) can act as an index into the other richer media stream (audio or video), which the user can play back for additional detail. Beyond searching, indexing, and playback, the Manga application demonstrates a process for automatically summarizing captured video as a way to help users locate high-level points of interest and then allows the user to drill down to specific points in the stream for playback [Uchihashi *et al.* 1999].

The variety of research and commercial investigations we presented indicates that the area of automated capture and access is a problem that has been and continues to be explored. In this dissertation, we identify a design process and an architectural framework that helps developers investigate the above issues in their explorations of the capture and access design space. Developers typically must address the minimal set of functional requirements of an application, as well as, those requirements not directly

related to the application features but they are highly relevant to the development of any ubiquitous computing system [Abowd 1999, Satyanarayanan 2001, Weiser 1993]. In overcoming these challenges, developers often may have little time and energy remaining to explore novel issues. Furthermore, developers usually have not leveraged existing systems as a platform on top of which they extend features. Specifically, the framework will address these problems and focus the design process on the essential features of the application—allowing the developers to better explore the design issues enumerated above.

CHAPTER 3

FORMULATING A GENERAL FRAMEWORK FOR DEVELOPING CAPTURE & ACCESS APPLICATIONS FROM LESSONS LEARNED IN PREVIOUS INVESTIGATIONS

In the previous chapter, we reviewed the existing body of related work. Although there are many examples of automated capture and access applications, these projects have explored only a small number of domains and issues. Unfortunately, those designers with creative ideas for automated capture and access often still face many challenges that prevent them from realizing their visions.

Since 1995, we have been performing our own investigation of the benefits of automated capture and access in a number of domains. In this chapter, we present a small set of the applications that we built to explore the capture of the classroom experience, highlighting the successes and failures of these development efforts. We discuss the challenges that we encountered while building these applications, as well as how we resolved them. Learning from these lessons, we hypothesize that a general solution can be engineered to abstract many of the issues we faced and remove them from the development of future capture and access applications. We then identify a novel architectural framework for designing capture and access applications

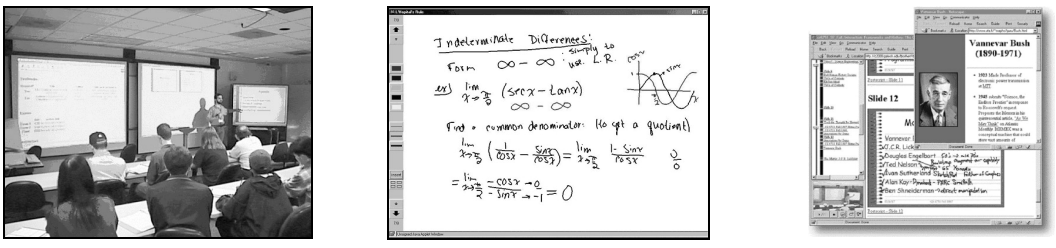
3.1 Challenges Encountered Investigating Classroom Capture & Access

In 1995, Abowd *et al.* began an experiment at Georgia Tech known as the Classroom 2000 project, later renamed eClass [Abowd 1999, Abowd *et al.* 1996]. The eClass project was an attempt to support both teaching and learning in a university

through the introduction of automated lecture capture. To examine the feasibility of this project, Abowd *et al.* initially developed a throw-away prototype, known as *ClassPad*, which we will not discuss in this chapter. Instead we will focus the discussion around the development efforts of an incremental/evolutionary prototype, known as the Zen* system, and the challenges we encountered as we attempted to add additional functionalities to it.

3.1.1 Zen*: Public Classroom Capture

In 1996, Brotherton *et al.* developed the Zen* system to study the effects of a classroom environment that captured details from the university lecture experience on behalf of the students and automatically generated a set of Web accessible notes immediately available after class for student review (see Figure 3-1) [Brotherton 2001, Brotherton *et al.* 1998]. To support this behavior, the Zen* system loosely integrated a collection of capture and access applications. Figure 3-2 displays the high-level architecture of the Zen* system and all its components. We now describe how the system evolved over time to include these various parts.



(a) (b) (c)

Figure 3-1. The original eClass. Figure 3-1a on the left shows a classroom augmented with capture. Figure 3-1b in the middle is a screenshot of the ZenPad application capturing an instructor's notes. Figure 3-1c is a web browser interface for viewing captured lectures. (Pictures courtesy of Jason A. Brotherton)

The Zen* system divided the task of capturing the various streams of information within the classroom among several specialized machines. In place of a traditional whiteboard, instructors used an electronic whiteboard (such as the LiveBoard or SmartBoard) that ran the *ZenPad* application, the main user capture component in the client-server Java-based Zen* system. This application automatically recorded the lecture slides presented in class as well as the instructor's handwriting. *ZenPad* included a very minimal interface to better support the degenerate case where the instructor only writes on the whiteboard by maximizing the actual screen surface for writing. Brotherton *et al.* configured a second machine inside the classroom to connect to the Internet through a proxy server, *ZenProxy*. This proxy server logs all URLs visited by browsers that go through it, providing a simple way to record all Web pages the instructor showed during lecture. Finally, a separate machine recorded the audio inside the classroom. A Java-based client application, known as *ZenStarter*, ran on this machine and automatically launched the RealEncoder program to record audio when a lecture begins. The *ZenStarter* application also terminated the RealEncoder application at the end of each lecture and then transferred the encoded audio to the server disk via FTP for later review by the students.

Each of the capture services inside the classroom connected to a central server, known as *ZenMaster*, which provided coordination for each lecture, or capture session. This coordination included the collection of prepared materials prior to a lecture, initiating and terminating the recording for all services for a given lecture, and the integration and post-production of all captured materials to create the Web-accessible notes. Overall, the Zen* system required some initial set up and maintenance, which

included the specification of all machines involved in the capture of a classroom before runtime. However, the eClass project succeeded largely because this coordination was transparent to the users, requiring very little extra instructor or student effort.

Over time, requests from users (both teachers and students) resulted in several nontrivial changes to the system. These changes included:

- *ZenViewer*, an extended whiteboard application that is a display surface showing the history of the lecture slides presented during class;
- Video capture; and
- A database of captured lectures to support additional access behaviors, such as a search function.

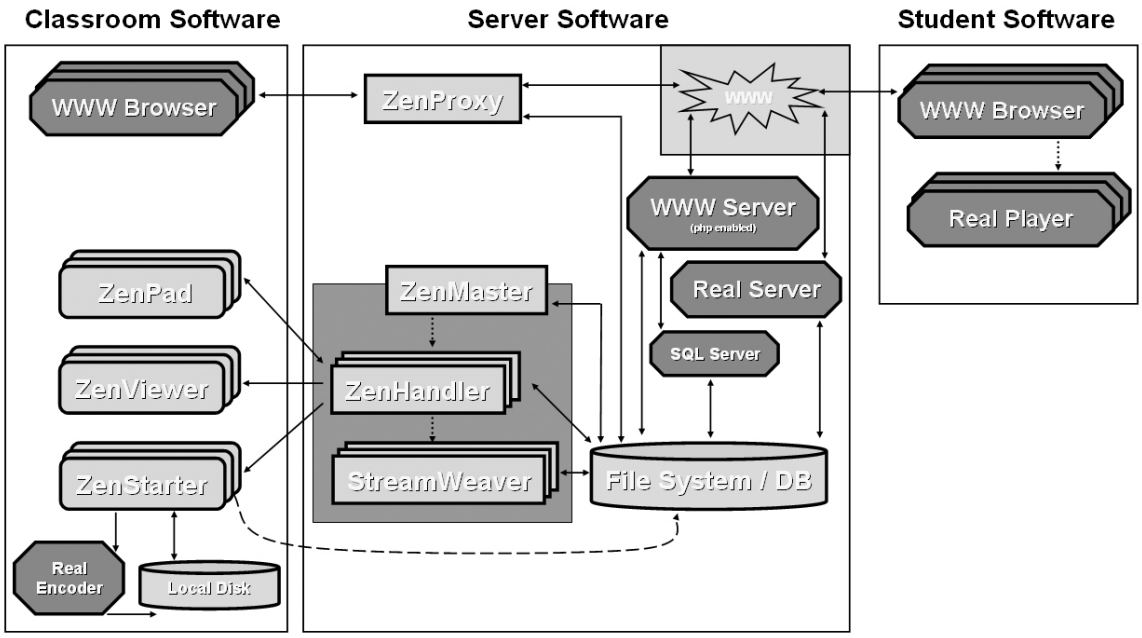


Figure 3-2. Components comprising the complete Zen* system. (Picture courtesy of Jason A. Brotherton)

This growth and change in the system, occurring over a 4-year period, was

possible largely because from very early on Abowd *et al.* adopted a structure to the capture problem that separated the different concerns into four phases:

- pre-production to prepare materials for a captured lecture;
- live recording to capture and timestamp all relevant streams;
- post-production to gather and temporally integrate all captured streams;

  and

- access to allow end-users to view the captured information.

The clear boundaries between these phases allowed the developers to evolve the prototype with minimal down-time to include the improved capabilities described above as small isolated changes to the software. Table 3-1 shows a summary of all the components in the Zen* system and the phases they support.

Table 3-1: Summary of Zen* components and the phases they support. Custom software is listed in bold and third party software is listed in italics. (Table courtesy of Jason A. Brotherton)

	Pre-Production	Live Capture	Integration	Access
Client	**Transfermation**	**ZenPad** **ZenViewer** **ZenStarter** *WWW Browser* *RealEncoder*		*WWW Browser*
Server	**ZenMaster**	**ZenMaster** **ZenProxy**	**StreamWeaver** *MySQL Server*	**PHP Scripts** *WWW Server* *Real Server* *MySQL Server*

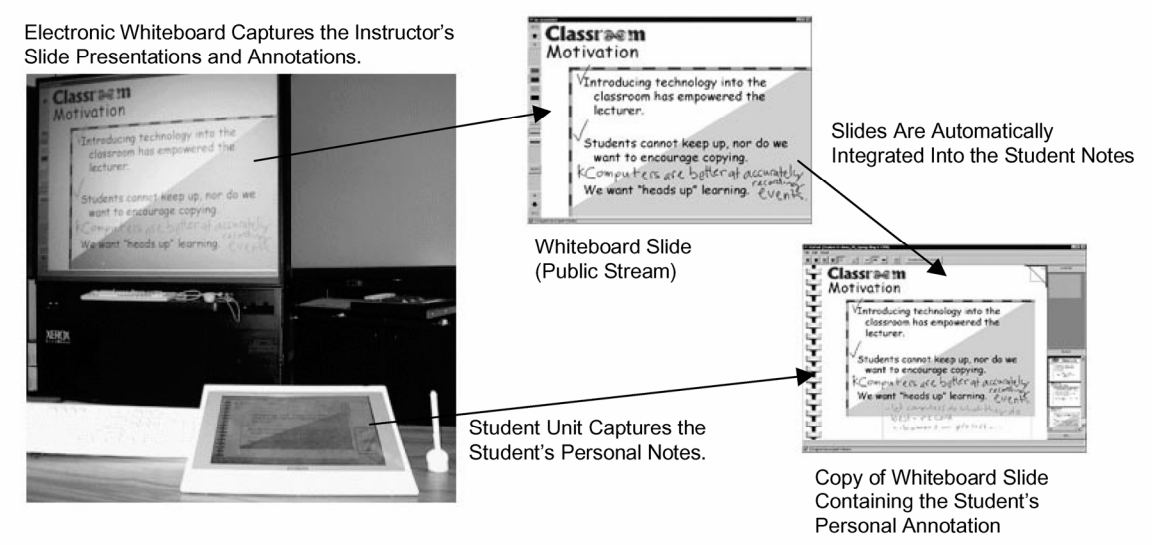


Figure 3-3. The StuPad system as used in classrooms supporting the public capture of the Zen* system.

3.1.2 StuPad (Student NotePad): Personalizing Public Classroom Capture

One goal Abowd *et al.* had with the eClass project was to relieve students from the need to tediously copy all the notes presented during class. However, the Zen* system only captured the instructor's actions during the lecture, excluding students from being able to make the captured record more personally meaningful. As a result, we noticed that some students continued to take small amounts of notes using pen and paper [Truong *et al.* 1999]. From surveys of students who used Classroom 2000 for at least one 10-week term, we learned that the students wanted to have their own notes, currently taken on paper, more tightly integrated with the public captured notes. In an open-ended question asking students to name the single feature they would like added to Classroom 2000, many indicated a desire for student note-taking devices. When asked explicitly, 62% (of 239 respondents) either strongly agreed or agreed (32% were

neutral) with the statement that the value of the captured lecture notes would increase if their personal notes were included. These results supplied the motivation for providing students with a way to electronically capture and integrate their personal notes with the public captured notes.

To better support the integration of each student's notes with the eClass notes, we developed the Student Notepad (*StuPad*) application as an extension to the existing eClass system. *StuPad* provided students with an interface that integrates the prepared presentation, digital ink annotations and Web pages browsed from the public classroom notes into each student's private notebook for additional personal annotations. We separated the design problem into two phases, capture and access, because students perform different tasks with different physical interfaces in those distinct phases.

Inside the classroom, we sought a reasonable complement of familiar paper-like

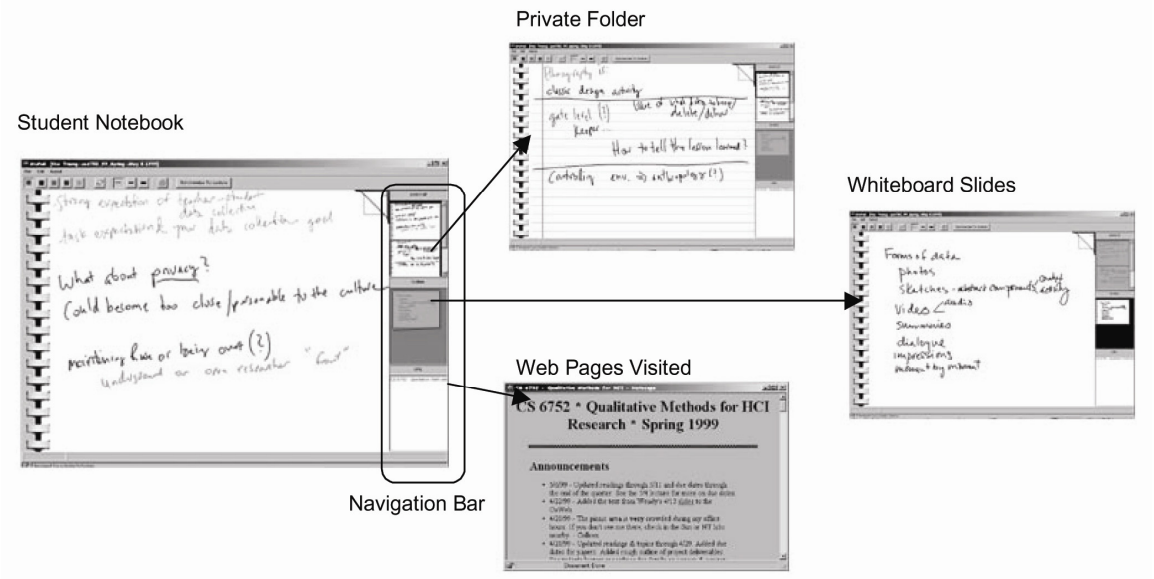


Figure 3-4. The StuPad capture interface.

functionality augmented by useful electronic functions. Because we had control over what devices to make available to the students, we designed *StuPad*'s capture interface to run on networked computers attached to pen-based video display tablets, as shown in Figure 3-3. The student's capture interface included separate sections for private note pages that the student alone controls, copies of whiteboard slides that the instructor is presenting (including the instructor's annotations) that the student can annotate, and Web pages that have been traversed by the instructor during class. *StuPad* provides a quick overview of the lecture activity through custom-built navigation bars located on the right side of the interface, which dynamically update thumbnail images of private notes and whiteboard slides. Tapping on a thumbnail of a page will load that page into the main canvas area in the center of the screen, where students can add personal annotations to a page of notes. Figure 3-4 shows how the navigation bar in the student interface allows for easy discrimination and switching between three streams, a private notebook, a combined personal/public lecture stream and a browsable Web stream. In the main canvas, the pen allows for digital ink annotations and simple navigation through "flicking" motions in the top-right corner (to advance a page) and along the left-hand binding (to go back a page). The Web navigation bar lists URLs visited by the instructor. Tapping on a URL will open a separate browser window to view (and navigate from) that Web page.

Outside the classroom, where it is not possible to assume that all students will have a similar input device and display, we designed *StuPad*'s access interface to run on networked computers with the more traditional keyboard/mouse interface. The access interface allows students to review the lecture experience by skimming through their

notes (which includes personalization of the public notes), to revise them, and the ability to index into specific points in the lecture experience for further review of audio or video (see Figure 3-5). Two adjacent panels display the student's and the instructor's perspective on the lecture. For any given time point during the lecture, the left panel shows what was on the main canvas of the student's unit and the right panel shows what was on the instructor's electronic whiteboard at the front of the room. A timeline at the bottom of the screen coordinates the display of the notes and the replay of a lecture. A "scrub" on the timeline shows the time during the lecture. Students can drag the scrub back and forth to advance both panels, and any active audio or video stream (displayed using a RealPlayer™ module from RealNetworks). All handwritten annotations for a given page are drawn on the canvas areas in light gray until the lecture time passes the time at which they were created and then they are redrawn in their actual color. Web pages visited during class are shown in a separate panel as a list; clicking on a URL opens up a separate browser window displaying that URL.

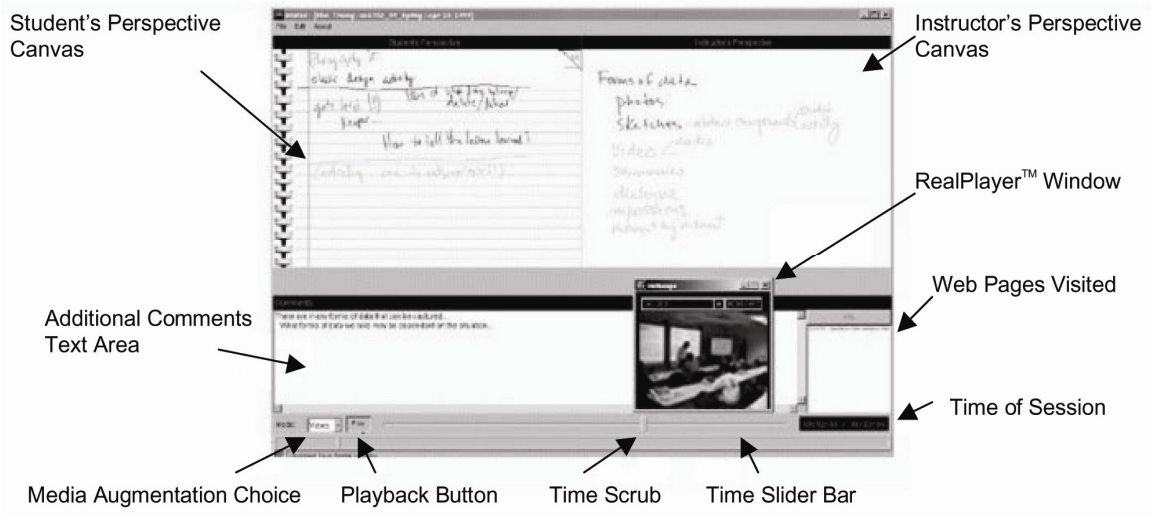


Figure 3-5. The StuPad access interface.

From a high-level implementation perspective, the requirements for student personalization of captured lecture notes meant that certain capture services had to be redistributed to a large number of clients. The instructor's whiteboard slides, annotations and Web pages visited are the public streams automatically incorporated into the student capture interface. Unfortunately, we had to work around the various manners in which the Zen* system collected and redistributed the different data streams. The existing communications structure of the Zen* system prevented us from implementing *StuPad* in the most obvious way, where each student notepad directly obtains the various information streams in the classroom.

The Zen* system already redistributes recent lecture slides to a neighboring *ZenViewer* extended whiteboard display to provide more persistence of the material presented during a live lecture. Ideally, each *StuPad* client would act as a *ZenViewer* client to receive captured slides and annotations. However, we were not able to leverage this feature without modifications. When multiple *ZenViewer* clients subscribed for captured information, problems in Zen*'s client-server network implementation were exposed. In Spring 1998, we performed a test to determine how many *ZenViewer* clients can connect to the Zen* server (*ZenMaster*) and subscribe for information captured by a single *ZenPad* capture client before the Zen* system noticeably performs poorly. We discovered that the client-server network implementation described above began to slow down the *ZenPad* capture client when there are more than 10 *ZenViewer* clients in the network.

Tracing through the communication scheme, we uncovered the cause of this

problem. After an instructor launches the *ZenPad* application to begin the capture of her lecture presentation, *ZenMaster* spawns a *ZenHandler* thread on the server side to coordinate the capture activities inside the classroom among the rest of the clients. As part of this coordination task, the *ZenHandler* thread immediately informs *ZenStarter* applications to begin capturing audio and/or video. *ZenHandler*'s primary responsibility then becomes receiving information captured by ZenPad and storing this information to the server disk. It also relays the captured slides and ink information to the *ZenViewer* client, which displays the recent slides and annotations. When an instructor writes on a slide, the *ZenPad* client sends the ink annotation to the *ZenHandler* thread on the server. This thread traverses through a list of *ZenViewer* clients subscribed for the information; it writes the ink annotation to each *ZenViewer* client before continuing to the next. A *ZenViewer* client completes its read of the ink only after it has finished receiving the information from the server and has rendered it. Likewise, a *ZenPad* client then completes its write only after the *ZenHandler* thread has finished writing the information to all the *ZenViewer* clients (and the *ZenViewer* clients each individually finished reading and rendering the information). In a single thread of execution, the server reads captured information from a whiteboard client, stores the data to disk and then writes the information back out to the *ZenViewer* client. On the client sides, networking and graphics operations both were handled in the same thread of execution as well. As a result, when many *ZenViewer* clients connected to the system, the *ZenPad* client's performance often bogged down and then eventually the system crashed because *ZenPad* waits for all of the *ZenViewer* clients to read a message while the instructor attempts to continue interacting with the whiteboard.

Despite our discovery of problems with the existing communication scheme, we did not make modifications to the Zen* system to accommodate the addition of the *StuPad* application. We did not want to risk adverse change to the working eClass prototype. However, because the instructor's slides and ink annotations were the primary streams of information for each lecture, we adopted a solution that extended the existing *ZenViewer* mechanism but removed portions of the code responsible for rendering the captured slides and ink annotations. Using this modified version of the *ZenViewer* client, we then created a separate multi-threaded server for the *StuPad* system to obtain all the material captured by the *ZenPad* client and redistribute these streams of information to each student's notebooks. This solution allowed multiple *StuPad* clients to connect with the Zen* without *ZenPad* needing to wait for all the clients to read a message and then eventually failing.

Web pages visited by the instructor during class formed the other stream of information that needed to be automatically integrated into the student capture interface. However, the redistribution of Web visits required a scheme that differed entirely from how we redistributed the instructor's slides and ink annotations. Unlike with ink and slide information, *ZenHandler*'s responsibilities do not include receiving and storing captured Web visits. At the end of the lecture, when the instructor terminates the *ZenPad* application, *ZenHandler* spawns the *StreamWeaver* application, which integrates and post-produces the access interface for student review. The *StreamWeaver* application must poll for Web pages visited during class in order to complete the post-production phase. The *ZenProxy* server does not communicate with the *ZenHandler* thread at all. This means Web pages visited during class are not immediately known

within the Zen* system and cannot be redistributed easily to the *StuPad* clients. Thus, to handle the redistribution of Web pages visited during class, we designed the *StuPad* server to periodically poll the *ZenProxy* application for the list of Web pages captured and computed what must be new Web visits since the previous list it obtained. The server then sends these new Web visits to the *StuPad* capture clients.

3.1.3 APE (Augmenting Public Experiences) with a CrossPad: Personal Capture beyond the Classroom

Despite the students' motivations to integrate their in-class personal notes with the eClass project's public capture, *StuPad* turned out to be a less useful application than we expected. When study occurred outside of class, additional notes taken by students remained difficult to integrate with the captured lecture notes. At that time, the IBM CrossPad presented an affordable solution that allowed one to work with pen and paper while also capturing an electronic record. Such a platform would have enabled students to capture notes inside or outside of the classroom.

Motivated by the vision of students using CrossPads to record all of their written notes, we designed a system (known as APE, Augmenting Public Experiences, with a CrossPad) to further augment the lecture experiences captured by the Zen* system. We imagined that inside of an eClass supported classroom, students could use the device to capture important points. When students review a lecture after class, they can take additional notes as well. And obviously, students can also use the CrossPad to record notes for other purposes. Because of the numerous capture situations, we envisioned the need to provide at least two ways through which students could review their notes captured with the CrossPad. The first method simply allowed students to access their

notes as a continuous set of pages recorded over time. By time-stamping all ink annotations and pages they create, students can later specify the date and time around when they wrote down information to retrieve the content they need. The system can also use simple temporal relationships, heuristics, or boundaries to automatically synchronize each student's personal notes taken during class to portions of the lecture captured by the Zen* system. This method of synchronizing a collection of student notes was used in the NotePals system [Landay *et al.* 1999]. Using a simple spatial-temporal clustering scheme, ink strokes can be aggregated into words and then sentences. During access, when a student studies a specific slide presented by the instructor during class, the Web interface for reviewing captured lectures automatically retrieves the pages of notes written by that user and displays it on the right side of the browser window. The student can select a particular page of notes to reveal the information she wrote down during class that are relevant to the current slide shown in the browser; the bottom of the browser renders the relevant ink clusters (see Figure 3-6). The student may arrange and organize the ink clusters, hide information she considers irrelevant to the current slide, and place important ink data over the slide as "sticky" content (meaning that even when the student clicks on another page of personal notes, these ink clusters over the slide remain).

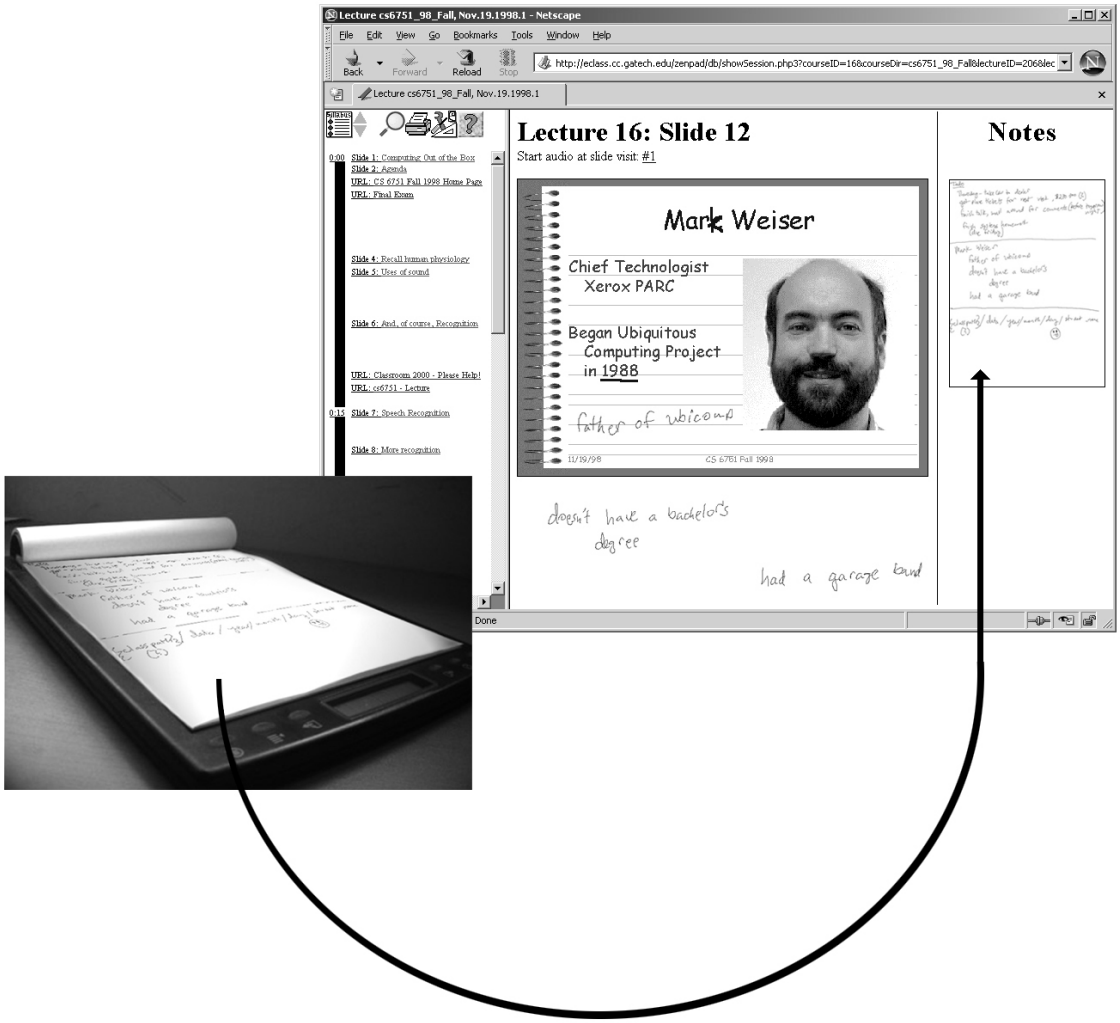


Figure 3-6. Our envisioned system for augmenting captured lecture experiences using a CrossPad. Pages of notes written on the CrossPad are later synchronized to slides presented during class. Individual ink strokes on a page are clustered. The access interface automatically displays those clusters that temporally align with the slide visited. Students can manually arrange the relevant notes or hide those that do not match semantically with the current slide.

Unfortunately, we faced many challenges integrating the system we envisioned above into the eClass project. First, the Zen* system stored captured information in a rigid hierarchical structure that consisted of course numbers, terms, and dates of the lectures. Information that students captured using the CrossPad, however, required a

more flexible organizational scheme. As a result, we did not think that forcing student's personal notes inside the same directory structure used by Zen* system to be a logical solution. Furthermore, the storage scheme employed in the eClass project suggested specific ways for accessing the captured lectures; students could identify the course and then the particular lecture date that they wish to review. Personal notes created during a lecture can be synchronized easily with the notes captured by the Zen* system, but notes created outside of a lecture, yet still pertaining to topics addressed in lecture, also needed to be integrated as well. As a result, providing flexible methods for reviewing the captured data proved to be a second difficult challenge. This form of access required the integration of the private notes and the classroom notes to happen through other contextual relationships beyond simple temporal synchronization. This issue also reaffirmed the inappropriateness of a rigid hierarchical storage scheme for a continuous stream of personal notes.

3.2 General Framework for Building Capture & Access Applications

We now use the lessons we learned from our experience with building the applications described above to form requirements for an architectural framework that supports the future development of capture and access applications. We use the lessons from the eClass project because:

1. Although our research group has looked at how the capture and access of experiences can assist people in a variety of other situations beyond college lectures (see [Truong 2001] for a review), we have encountered design and implementation hurdles in these other projects similar to

those we encountered in the classroom systems described in the previous section.

2. We experienced relative success with the classroom research because we could evolve the system over a long period of evaluation; this was aided considerably by the key architectural decision to structure the system into four phases (pre-production, live capture, post-production/integration, and access), allowing developers to easily modify various applications in the software system in isolation, without risking adverse changes to the rest of the system.

3.2.1 General Separation of Concerns for Capture & Access

Abowd *et al.*'s decision at the beginning of the design process to structure eClass into four separate phases facilitated much of the early evolution and extension of the system. As a result, we extend upon this design concept to identify a general set of concerns for capture and access applications. From our experience investigating automated capture and access of experiences in other domains, we now believe it is more appropriate to separate an application based on four different functional concerns instead of the four phases of operation used in the eClass project.

First, if an instructor prepares lecture material that she uses during lecture, only then does the pre-production phase occur first. On the other hand, the capture of information always occurs, producing information that is post-produced immediately after each lecture to generate Web pages that users may access later. While the pre-production phase was accounted for in the eClass system; we argue that in actuality, this

step exists outside the scope of the actual live lecture experience. We do not consider pre-production to be a generalizable concern, because most capture and access applications typically focus only on supporting the recording of live experiences and providing future access to that information. Furthermore, although the pre-production phase may exist for the lecture experience, definite pre-production phases may not exist for every live experience. We believe this furthers the argument that the pre-production phase does not generalize well to other applications.

The phases used in the design of the eClass project also imply a sequential ordering of activities, but it is better to consider the functional components of the overall architectural solution. As our exploration of capture and access continued beyond the classroom, we learned several other important architectural lessons:

- Information must be stored (or persist) until it is later accessed. This may seem obvious, and is implied in the post-production/integration phases of eClass, but it is often overlooked as a design issue. Upon encountering difficulty storing student's continuous streams of personal notes into the existing hierarchical storage used in the Zen* system, we realized that the lifetime of an application depends heavily upon the flexibility of its information storage model.
- The post-production/integration activity is further separated into concerns pertaining to: storage, as discussed above; transduction (or transformation) into different data types; and integration, in which relationships between separately captured streams cause the multiple streams to be delivered collectively during access.

- Access happens on varying time-scales, depending on when information is accessed relative to when it was captured. Therefore, different forms of access interfaces and services are desired, such as a summarized view that allows the user to drill down over summaries of long-term captured data.
- Information integration occurs due to different contextual relationships between captured streams. Examples are streams captured at the same place and time, or covering the same topic, or captured by the same person. Additionally, various pieces of information captured over time may be related [Pimentel *et al.* 2000, Pimentel *et al.* 2001]. As a result, we believe it is most appropriate to support the integration of information dynamically as a part of access.

Therefore, we have capture, storage, transduction and access as important building blocks for all capture and access applications, with no implied ordering of when they occur relative to each other. The integration of information should be performed when information is being accessed.

3.2.2 Decoupled Communication Structure from Essential Application Features

Additionally, although the software structure designed into the Zen* system facilitated much of the early evolution and extension of the system, we saw later that this structure affected the ability to scale up the system, as seen with *StuPad*. The structuring of eClass into four phases specifically resulted in extensions to the system that introduced inconsistencies between how the eClass server (*ZenMaster*) communicated with the different clients. These inconsistencies further weakened a

communications scheme that already had performance problems due to its lack of multithreading.

Brotherton *et al.* also tightly coupled this communication scheme with the essential application features in the Zen* system. This tight coupling meant that the communication scheme cut across all phases of the system. As a result, despite discovering scalability problems in the way the eClass project was designed to communicate over the network, modification of this aspect of the system would have required a global redesign across all four phases. Such a change would have been too risky, and as a result, we deferred to the makeshift solution of the StuPad server which respected the original eClass design and connectivity assumptions. However, this issue could have been avoided if the essential application features had been decoupled from this concern, making the system much easier to build and extend.

By decoupling the underlying communication structure from the application, a network abstraction can be introduced into the development of future capture and access applications. Although a few capture applications operate on a single stand-alone device, most systems are built using multiple networked devices. These devices need to be coordinated to communicate and share information. This issue can be resolved through any number of client-server or peer-peer configurations; however, this issue is not directly essential to the application design. Instead, when network communication is abstracted away from this architecture, a system with components spread over the network can be developed the same way as if all the components reside on a self-contained device.

3.3 Focused Development Process

The key features of the architectural framework described in the previous section suggest a general software structure for capture and access applications. We used this software structure to identify a user-centered design process that allows developers to rapidly prototype applications to investigate the usability and usefulness of novel capture and access ideas. This design process focuses the design task on two types of issues:

- ***Information design,*** which involves identifying, structuring and presenting the information that the user interacts with during the capture and access phases; and
- ***Interaction design,*** which involves designing interfaces and techniques to support the communication between the user and the application during the capture and access phases.

This *focused* design process encourages developers to specifically consider the two types of issues described above as they tailor the four core architectural concerns into a specific solution for a given domain problem.

3.3.1 Design the Access Interface

The point of this class of ubiquitous computing applications is the eventual *access* of the captured information. As a result, we urge developers to let the access needs of the end-user drive the design of the application. The steps involved in building the access component in a capture system are:

1. *Identifying what the user will want to review during the access phase.* In order to determine how the access interface will look and what it will show, the developers must study the application domain to gain an understanding of the user's needs.
2. *Determining how to present the captured information to the user during review.* The user may want a quick summary of the captured content and/or the ability to replay details. Again, the developers must determine the user's needs and create the interface that supports the appropriate user interaction.
3. *Specifying how the access interface will query for the captured information from other components.* This query must contain enough detail to allow the capture system to discriminate between the information that the user wants to review from the remaining set of previously captured content.

By designing the access interface first, developers can minimize the amount of effort involved in developing components may capture information that may never be reviewed by the user. This step also helps to preserve the user's privacy (and the privacy of others involved during the capture phase) by not capturing more than what the user will need later.

3.3.2 Design the Capture Interface

Once the access requirements have been addressed, developers can then assess

what capture components they need to create. The steps involved in building the capture interface are:

1. *Identifying what information to capture during the live experience.* To provide the access interface with the information the user wants to review, the developers must determine what can be captured during the live experience to result in the desired content during access. The raw captured content can differ from what the access interface requires, but it must be information that can be processed into what the user needs later.
2. *Determining how to capture the information.* Capture must happen unobtrusively. The developers must devise methods that naturally or implicitly preserve information from the live experience without disrupting the user from her activity.
3. *Specifying how to tag the captured information to allow other components to obtain it.* The application must annotate the captured information in such a way that distinguishes content from one captured session from the next. Developers can use their knowledge of the queries that need to be supported during the access phase to determine the set of meta-data tags to use.

As part of the design process, developers must resolve trade-offs between what the access application requires and what information can be captured naturally and/or implicitly from the live experience. Developers can iterate upon the design of the access interface and that of the capture interface until they reach a compromise that captures

enough to allow the user to review what she needs.

3.3.3 Design the Storage Components

Next the developers should determine how to store the captured information. The steps involved in building the storage components are:

1. *Determining how to store information.* Storage can be tightly coupled with the capture component or it can be supported elsewhere on the network. It can also exist in a single location or distributed across multiple locations. For example, an application that captures information for multiple users may allow different users to store personal information on devices that only they can access. The application developers must decide the proper storage model for their specific system. Once the appropriate model has been determined, developers must then design the back-end support for each storage component.
2. *Specifying how the storage component will query and/or subscribe for information from other components.* This query must contain enough detail to allow the capture system to discriminate between the information that the storage component can store from the remaining set of captured content.
3. *Identifying what information a storage component can provide (in reaction to queries).* Based on what the storage component stores, it must be able to return requested content to a component. However, developers can determine what queries a storage component supports or not.

Although this is an important architectural concern, this aspect of the system can be abstracted through a reusable component that provides generic storage support. Unless an application calls for custom storage behaviors, developers can skip this portion of the design process.

3.3.4 Design the Necessary Transducers

Finally, developers must build transducers to transform and transcode information to ensure that the access interface can obtain all the specific content that the user wants to review This includes changing data formats (*e.g.,* wav files to mp3 files) as well as data types (*e.g.,* raw audio to text transcripts). Developers must iterate upon the design of the entire system to support transduced information as it would support other captured information. The steps involved in building transducers are:

1. *Identifying if there are mismatches between information used by access and storage components with what is captured.* Because the captured content can differ from what the access interface requires and what the storage component supports, the information must be processed. Developers must determine what mismatches exist.
2. *Determining how to transduce the captured information to yield the necessary content.* Once the developers have determined what captured content can be transduced to result in the information that the access interface requires and/or the storage component preserves, developers must then design the back-end support for converting between these data

formats and types.

3. *Specifying how the transducer will query for the captured information from other components.* This query must contain enough detail to allow the system to discriminate between the information that the transducer can process from the remaining set of captured content.
4. *Specifying how to tag the transduced information to allow other components to obtain it.* The application must annotate the transduced information in such a way that can satisfy queries by the access interface and storage components.

3.4 Summary

In this chapter, we presented our experience with building a small set of the capture and access applications our research group has developed and studied between 1995 and 2001. Specifically, we discussed the issues we encountered exploring the benefits of capture and access in the classroom domain. Although we ultimately created a solution to these challenges, we spent a significant amount of time and effort that could have been focused on exploring and solving human-computer interaction level issues. The specific systems level challenges we encountered were caused by the fact that:

1. Capture and access applications are built as an *ad hoc* confederation of components. Developers can not determine *all* of the features of a system at design time. As seen with the Zen* system, capture and access applications must be extended to include devices and components as

additional functionalities become necessary.

2. Not only can developers not pre-determine all of the functionality for a capture and access application, but perhaps more importantly, they can not pre-determine all of the types of captured information, because applications are comprised of an *ad hoc* confederation heterogeneous devices and components. As a result, a rigid hierarchical storage scheme must be modified and made more flexible.
3. Common architectural concerns exist across applications but were often rebuilt for each system. These concerns can be abstracted as reusable components to minimize time and effort spent reengineering these solutions.
4. Application code is often tightly coupled with the underlying network communication scheme. We can not easily modify the application code nor the communication scheme without risking adverse changes to the other. However, the design of a software application must be iterated upon during its design, development and deployment phases.

We used lessons learned from addressing these challenges to form the basis for a generalized high-level architecture for capture and access applications. We derived capture, storage, transduction and access as important building blocks for all capture and access applications, with no implied ordering of when they occur relative to each other. Furthermore, we believe the integration of information should be performed when information is being accessed. This framework decouples the underlying

communication structure from the essential application features, effectively abstracting networking concerns from the development process. We then used key features of the architectural framework to identify a *focused* design process that abstracts common, accidental development tasks and allows developers to address the interaction design and the information design issues specific to a software application. In the next chapter, we describe the implementation of an infrastructure that embodies this architectural framework and the accompanying design process.

CHAPTER 4

THE IMPLEMENTATION OF THE INCA (INFRASTRUCTURE FOR CAPTURE & ACCESS) TOOLKIT

In the previous chapter, we discussed the challenges we encountered in developing capture and access applications. To overcome these issues, we needed to devote a substantial amount of time and effort addressing these challenges—resources that could have been spent better on the design and refinement of the applications themselves. In this chapter, we discuss our application of the architectural insights we learned from the experiences previously described towards the design of an infrastructure/toolkit, known as INCA (Infrastructure for Capture and Access) [Truong and Abowd 2004].

We developed the INCA toolkit primarily in Java. The infrastructure consists of 49 core classes and approximately 13,721 lines of code. As part of the package, we also release a number of custom components that provide specific functions. These additions to the toolkit include another 43 object classes that consist of 8,001 lines of code (at the time of this publication).

4.1 High-Level Features of the INCA Toolkit

INCA embodies key design abstractions and a separation of concerns to lower the barriers for developing and evolving capture and access applications. Features of the toolkit aimed to facilitate the development of capture and access applications include:

- A collection of reusable, distributed modules that support the capture, storage, transduction, and access of streams of information. These modules allow

designers to divide complex software systems into smaller parts that perform specific functions, making the application easier to build and to manage.

- A data-centric application design model that supports the attribute-based storage, retrieval, and garbage collection of data. The toolkit includes an extensible component that automatically attaches context and metadata tags to captured content. This scheme enables the development of applications designed for a variety of different domains; it also allows INCA to handle many different kinds of data homogeneously.
- Additional components for observing and controlling the run-time state of the system. Together, these modules support the dynamic adaptation of application features and also allow developers to implement privacy and security features.
- A network abstraction layer that hides details about the underlying communication scheme used by the distributed modules. This feature decouples the underlying communications scheme of a system from the actual application code and allows designers to focus on the essential features of an application and the interface.

4.1.1 Separating Common Architectural Concerns

Using the architectural insights we identified in the previous chapter, we generalized a list of architectural similarities common across this class of applications. We then applied the basic separation of concerns principle towards the design of the INCA toolkit to encourage any capture and access application to be expressed in terms of these basic functions:

- Part of the system is responsible for the **capture** of information as streams of data that are tagged with relevant contextual metadata attributes.
- Part of the system is responsible for the **storage** of attribute-tagged information.
- When information needs to be converted into different formats and types, part of the system must **transduce** the information.
- Part of the system is responsible for the **access** to multiple, related streams of information that are gathered as response to context-based queries, *i.e.,* support for the integration of information can be wrapped directly into support for the

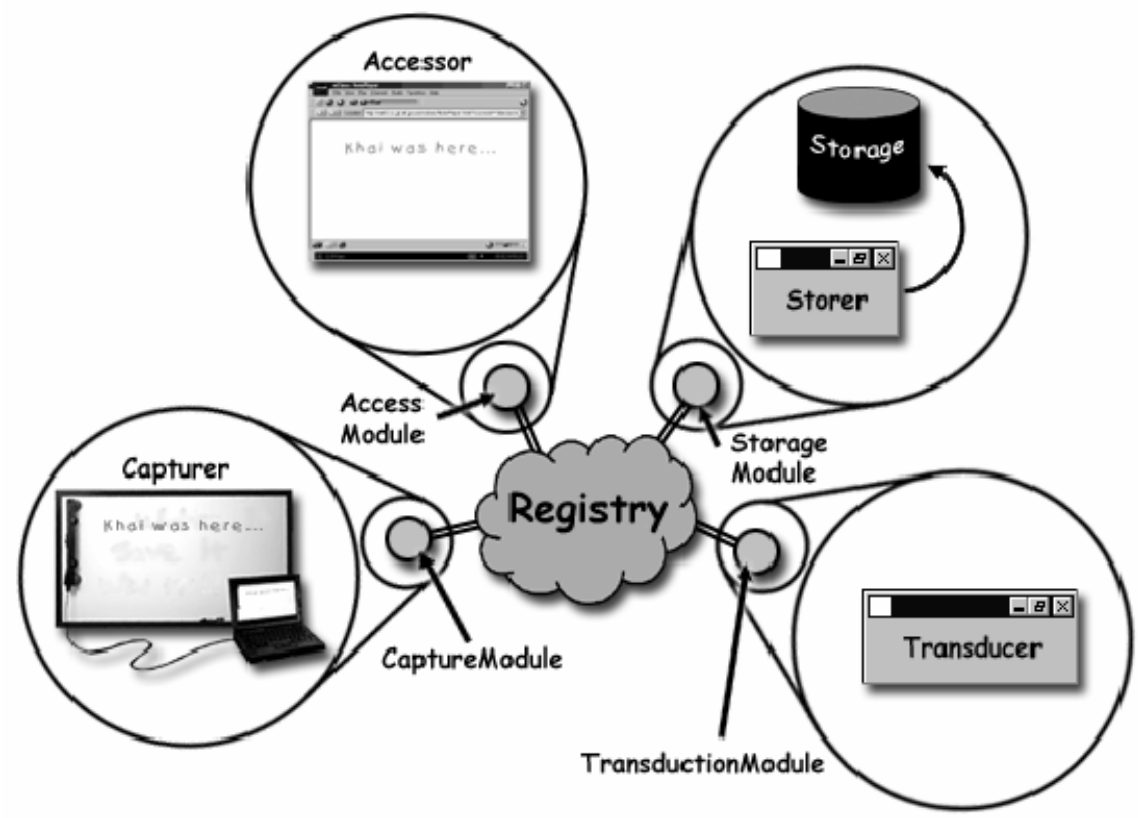


Figure 4-1. General architecture for systems built using INCA. A *Registry* runs at some well-known location and any number of applications acting as *Capturers, Accessors, Storers,* or *Transducers* can connect to it and share captured information through instances of specialized networked modules (such as a *CaptureModule, AccessModule, etc.*). This diagram only shows the core functional modules. Other components not shown include: *GarbageCollectors, Observers, Controllers, Taggers* and *Queriers*.

access of the information, such that when information is requested, related streams of information are jointly provided.

For any given application, there may be more than one instance of each of the above functions. From an implementation perspective, INCA provides a direct way to translate applications designed in the above manner into an executable form. The toolkit includes a number of modules that a programmer extends or uses as part of the application code, see Figure 4-1.

As we discussed in Chapter 3, developers may decompose specific applications into a collection of capture, access, storage and transduction components, divided into phases with an implied sequential ordering of activities. However, this ordering differs across different applications. To facilitate the development of a variety of applications, we provide these functional components of the overall architectural solution as building blocks with no implied ordering of when they occur relative to each other.

4.1.2 Supporting a Data-Centric Application Design Model

In Vannevar Bush's description of the memex [Bush 1945], he describes the user being able to create associations between different artifacts so that when she requests information about a topic, the system returns all the relevant content to the user. As previous applications have strived to support capabilities similar to the mythical memex, each has explored storing and integrating information streams using various contextual relationships, such as time. This means that applications rely heavily on the metadata of the captured information. Thus, from the early stages of this work,

we made the design decision to incorporate the use of attributes, or metadata, into the way that INCA captures, manages and retrieves information.

In making this decision, we allow INCA to support the integration of information within its support for the access of the information, such that when an application requests information, related streams of information are jointly provided. Additionally, developers often re-implement methods for capturing, storing, and retrieving different data types to meet the specific needs of the application. This process typically results in an efficient design and a well-constructed application; however, this same structure also makes systems hard to evolve. For example, storage schemas appropriate for a pre-determined situation may need to be altered when users later recommend additions to a system. Likewise, network communication protocols suffer from the same issue. This problem exists largely because developers have created specific support for each type of captured data.

INCA treats captured data as large binary objects, known as *DataObjects*. To provide applications with knowledge about the captured data, *DataObjects* contain, in addition to the byte array that holds the actual content, a list of *Attributes* that describe meta-information about the data. This homogeneous representation of captured content abstracts specific details about the information from the infrastructure. This feature means a generic communication scheme can be developed and used for different applications. Likewise, INCA also provides generic support for capturing, storing, and accessing the captured information streams. The specifics on how to manipulate and use the captured information become domain/situation design issues that the developer resolves for each application. For the infrastructure to be able to handle the captured

data, the meta-information, or the *Attributes* that describe each piece of information, plays an important role.

Application developers must consider carefully how to capture and tag information in a manner that allows access applications to retrieve the specific piece of information at a later time. As a result, developers should think about the access behaviors they want to support and to tag captured content with attributes that match those types of queries. To facilitate this process, the toolkit includes an extensible *Tagger* component that automatically attaches context and meta-data tags to captured content that developers can instantiate or customize within their own applications. INCA also provides an extensible *Querier* component to assist in specifying requests for information. This data-centric application design model also supports the attribute-based storage and garbage collection of data.

4.1.3 Observing and Controlling the Run-Time State

In addition to all the above features that make the development task easier, provided developers with the ability to allow various application stakeholders to observe and to control the run-time state of a capture and access system. These capabilities together support a number of interesting features, including:

- *The dynamic adaptation of system features.* In being able to know the state and availability of various capture and access components on the network, application developers can build context-aware applications that react appropriately to the set of services present in the environment. For example, if an application knows that a meeting will soon start in a

specific room, it can identify and activate all of the capture components located in that space.

- *The protection of privacy through the identification and control of capture and access services.* An inherent concern for privacy often arises from automated capture because it may involve the preservation of potentially important and personal information. We begin to alleviate this concern by providing a way for users to understand the state of the system. Application developers can integrate this capability into their systems to allow users to know what the environment captures. This feature enables the user to adapt her behaviors accordingly. Additionally, the user can turn off capture during a live experience to keep aspects of the experience private.

The privacy problem, however, extends well beyond just simply making people aware of capture. Issues such as how to limit / control access to captured information and how to securely store, retrieve and share captured content must also be addressed. We argue that this problem is by itself a research problem that should be left for experts to explore. We enable these explorations by providing the INCA Toolkit as a platform above which researchers and developers can investigate different techniques to deal with the privacy and security concerns. For example, in Section 6.6, we present a token-based access control mechanism added to INCA by a Computer Science Ph.D. student to consistently address this concern in all future applications developed using the toolkit [Iachello,

2005]. Although the researcher did not want the access control scheme to rely on encryption, this could have been added to the key-lock scheme, where the modified toolkit encrypts the captured data using the lock and decrypts the data using the key. Encryption is important feature that needs to be considered as an additional way to protect privacy via the access software.

- *The evaluation of system use.* Evaluators can use the ability to observe the run-time state of a system for various purposes, such as being able to inspect and log how users interact with the system. This allows evaluators to learn when the user interacts with various capture and access components, as well as what information the users create or review. Additionally, this feature can track when the system adds or removes components; for example, this can happen when the user introduces a new device to an environment or when a developer adds new functionality to an existing application.

4.1.4 Abstracting Underlying Network Concerns

As seen with eClass, the communications structure of a system often cuts across all the architectural concerns of the capture and access application and plays an important factor in evolving these applications. We include a network abstraction layer to separate the underlying communications concerns from the application code (see Figure 4-2). This layer supports a general client-server architecture. We build all the functional modules in the toolkit (that support the capture, access, storage, transduction

of information and the control and observation other components) as clients in this architecture and a *Registry* object as a server that maintains a list of the available functional modules.

To support a variety of applications designed for different domains, the server and the clients only exchange serialized message objects. By viewing captured data as only raw bytes with tagged attributes, the infrastructure is able to handle any number of different kinds of data in the same generic fashion. As a result, this network layer coupled with the strong use of attributes in the capture, storage, and access of information enabled us to build the functional capture and access modules in a manner that allows application developers to create applications without needing to know how the different parts communicate with one another. As a result, the developer can write code for a complex distributed system that involves multiple devices in the same

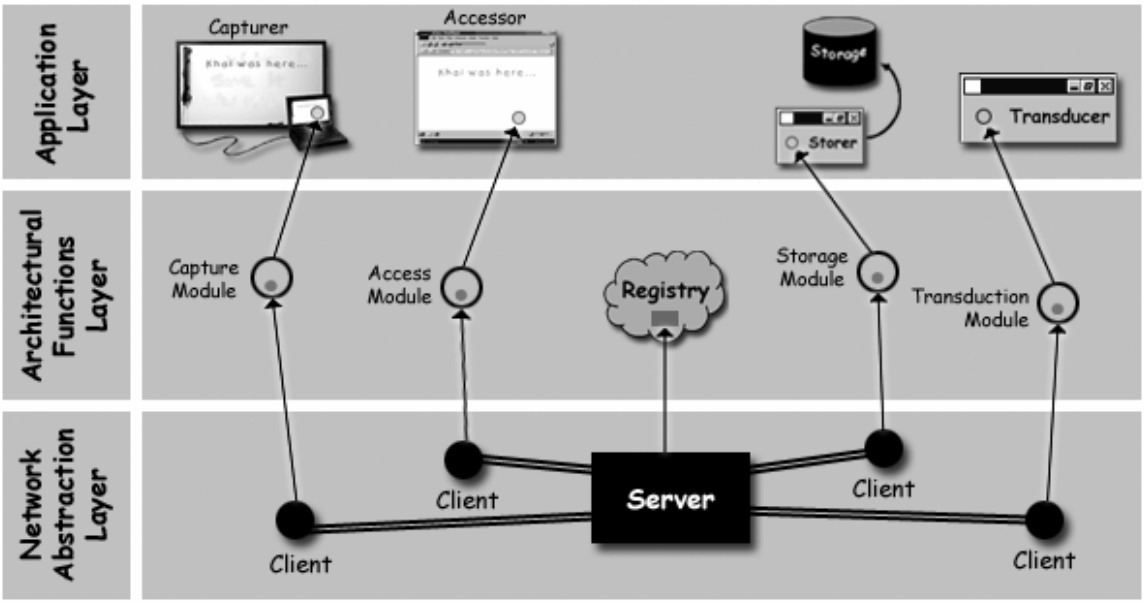


Figure 4-2. Software layers supported by the INCA Toolkit. INCA includes a network abstraction layer on which the specialized networked modules (such as the *CaptureModule, AccessModule, etc*.) and the *Registry* are built. Developers create applications without worrying about details of the underlying network code.

manner as she would for an application that runs on a single device, without any knowledge of the inner workings of INCA. We provide additional detail on our specific implementation of the network abstraction layer in Appendix A.1.

4.2 Reusable Architectural Functions Available to Developers

In addition to a network abstraction layer, the high-level features described above require toolkit support for the following architectural functions:

- Capturing and tagging information
- Storing information
- Accessing information
- Transducing information
- Observing the run-time state of the system
- Controlling the run-time state of the system.

We built all these functions as reusable building blocks that application developers can customize to meet the specific requirements of their system. In this section, we present the toolkit support for each architectural function and how developers would use these reusable components. We provide additional detail on our specific implementation of these reusable objects in Appendix A.2.

4.2.1 Capturing and Tagging Information

INCA defines a *CaptureModule* object to support the capture of information, where capture is defined as the act of collecting data from the physical environment. The `capture` function defined in the *CaptureModule* is invoked when the application

attached to a device, such as a camera, microphone, or electronic whiteboard, has data that is available to be tagged and stored, transduced or provided to some access service.

Data is captured and digitized as *DataObjects*—Java objects containing raw bytes and attributes that describe some properties about the data (such as its data type or format) and the context of the captured activity (such as when and where it was captured). These tags are used in later stages to make the captured data automatically available to those parts of the system that are responsible for storing, transducing, or otherwise accessing it.

A *CaptureModule* can register to various *Tagger* objects to add metadata information automatically to the output objects from the capture function. INCA provides a number of reusable *Tagger* objects in its toolkit library for adding attributes specifying the location of the captured activity, the people present in the location, the current time, and the data type being captured. Developers can create their own *Tagger* objects as well by implementing a `getAttributes` callback function that receives a *DataObject* as a parameter. This function returns a list of attributes (or an *AttributeVector*) that it has computed as additional properties that should be added as tags to the specified *DataObject*. The *CaptureModule* object will then merge this list of attributes with the current list of attributes for that *DataObject*. The *CaptureModule* omits attributes already present in the current list of attributes to prevent the addition of redundant meta-data.

We provide a sample implementation of a *Tagger* object in Appendix B. This *PeoplePresentTagger* object associates captured data with the names of people present in a specified location. We highlight the minimal lines of code needed to extend the

Tagger object with custom behavior. The remaining lines of code generate the specific attributes to be associated to the captured content.

The *PeoplePresentTagger* object illustrates a sophisticated tagging operation that involves the *Tagger* object communicating with another application (in this example, it uses the *SignPostSubscriber* object to read context from the Context Toolkit) to subscribe for changes people's location within a house. The *PeoplePresentTagger* object uses the set of changes in people's location to determine the list of users present in a particular location and returns this information within an *AttributeVector.*

A simple *Tagger* object, such as one that returns time stamps, can simply create an *Attribute* that holds a time value and return it within an *AttributeVector* container object. A slightly more complex *Tagger* object, such as a one that tags captured audio with the noise level, would process the *DataObject* to obtain the amplitude of the audio signal that it returns as an attribute.

4.2.2 Storing & Managing Information

Automated capture often results in the archival of unwieldy amounts of data. To store and organize information, the INCA toolkit contains a reusable repository component that a developer instantiates or extends in their own applications. Additionally, we provide a mechanism for selectively discarding content as a compromise between recording only those things predetermined to be important and recording everything.

A *StorageModule* provides persistence for captured data. A *StorageModule* can

specify a list of attributes for the kind of captured information it is interested in automatically receiving via a `subscribe` function. When the capture function in a *CaptureModule* is invoked, the `store` callback function of any *StorageModule* that has registered a satisfied set of attributes is provided with the captured data. When access to stored information is needed, the `retrieve` function is called. How this information is actually stored and retrieved is left up to the part of the application that actually extends the *StorageModule*.

This attribute-based storage approach allows developers to create applications that rely on a single repository that stores everything just as easily as a scheme that employs a number of repositories, each storage storing specific content. The appropriateness of one design over another depends on the actual design situation.

A *Repository* service included in INCA defines all the basic *StorageModule* functionality. It provides a relational database and supports the storage and retrieval of any kind of data tagged with attributes. A *Repository* can be launched and left running, so that application developers can have storage performed as an existing service without additional development effort or modification. The *Repository* class can also be extended to meet a specific application need, such as storing only personal information or optimized for a specific captured data type. By default, The *Repository* provided by INCA uses MYSQL as the back-end database. The decision to use a MYSQL database as the back-end storage component does lead to some deployment issues; mainly, requiring the availability or installation of the MYSQL server. As a result, we later modified the *Repository* to use 100% Java database back-ends, such as PointBase™ and HSQLDB.

We have also implemented a *FileRepository* service which provides the exact functionality using a hierarchical file structure. The *FileRepository* uses three files that hold essentially the same information as the three tables used in our SQL implementation. Instead of storing the *DataObjects* within the file that acts as the data table, this file instead stores the locations of the *DataObjects* in the file system. The *FileRepository* actually stores the *DataObjects* on disk and uses folders to manage the exploding number of files in a single folder, a problem that can affect the time to list and fetch information from disk. The folder structure consists of the year, month, date, hour and minute at which the data was captured and timestamped.

INCA provides a *GarbageCollectionModule* object to allow an application to perform the attribute-triggered discarding of unwanted information. Similar to how information desired for review can be specified using a query over the attribute tags, a designer can invoke the *GarbageCollectionModule*'s `gc` function by specifying attributes of data she wants storage repositories to remove from persistence. In the current implementation, storage repositories actually delete the data, meaning that the discarded content can never be recovered after that point.

For situations such as when a user writes an ink stroke and then erased it, yet the review session should show that the user had written the ink and later erased it, then this form of deletion should be handled differently. The *CaptureModule* includes an `update` function that takes in an existing data object and updates it with the new information. We recommend appending a "deleted_at" attribute where the value is the time at which the user erased the ink stroke. This allows the ink to still persist in the system and allows the application to decide how to render the information (based on if

the “deleted_at” attribute exists for that piece of captured information).

4.2.3 Accessing Information

The INCA Toolkit includes an *AccessModule* that can be used by application to obtain information for users to review. An access application can use the `subscribe` function to receive data as it is being captured (*e.g.*, subscribe for all data created by “John” originating from “Building 4: Room 106”). When the capture application records new information, INCA automatically invokes the *AccessModule*’s `handle` callback function to provide the access interface with this data.

An access application can also perform a context-based query for information that has been previously captured using a `request` function. Upon receiving this query, INCA checks with all existing *StorageModule* and *Repository* instances for data matching the specified query (by invoking the `retrieve` function in the storage components) to obtain all matching data, resolves any cases of redundancy and then returns a list of data found back to the requesting *AccessModule* object.

As mentioned previously, the support for the integration of information is wrapped into the support for the access of the information. Information is integrated based on how it is requested through context-based queries. In its simplest form, the query match is based on attribute name-value pairs and can grow to include more general data retrieval operations that more effectively filter and mine large distributed repositories. Various temporal and spatial integration techniques have been explored in the past. These techniques, and more, can now be created as reusable integration services.

One example of how INCA simplifies the programming task is seen in the relationship between an *AccessModule* and the rest of the run-time system. An *AccessModule* makes context-based queries for information (*e.g.*, deliver all data owned by "John" originating from "CRB 381" in the past week) but the application programmer does not need to know where any of this captured data resides. The run-time system of INCA automatically resolves the query and delivers the information to the requesting *AccessModule*. This feature allows the programmer to focus on high level issues such as how to allow the user to specify and interact with the information she wants to review. However, as the case study reported in Section 6.9.3 revealed, the developer of the access interface must still have an understanding of what information has been captured and how it was tagged previously.

Although we did not implement this feature originally, we ultimately realized that an *AccessModule* can register *Querier* objects that request specific information in a similar manner to how a *CaptureModule* registers *Tagger* objects that automatically add metadata information to captured data. A developer simply defines the `getQuery` method of the *Querier* object to specify a query for the desired content. We envision that these components can be reused across applications to save the amount of effort needed to redefine user interface components to obtain parameters that describe the information that the user wants to review.

We provide a sample implementation of a *Querier* object in Appendix C. This *TimeQuerier* object generates time parameters specifying what information to retrieve (see Figure 4-3). We demonstrate that this object can be an interactive GUI component that exists as part of the access interface. This example illustrates a sophisticated

Querier object that acquires a user defined query, where the user inputs time parameters. We highlight the minimal lines of code needed to extend the *Querier* object with custom behavior. The remaining lines of code create the GUI for the *TimeQuerier* component shown in Figure 4-3. Depending on how the developer intends to obtain the time parameters, the number of additional lines of code, such as those shown for the GUI, will differ.

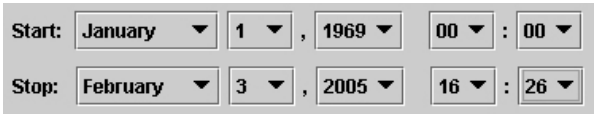


Figure 4-3. Simple GUI for TimeQuerier object. This interface allows users to specify the start and stop time of the information she wants to access.

An *AccessModule* will invoke the `getQuery` function of each registered *Querier* object. The *AccessModule* performs an `AND` operation over all the collected queries and returns to the parent application the resulting information retrieved using this final query.

4.2.4 Transducing Information

A *TransductionModule* supports the transformation of information between different data types (such as from a video file to a series of image frames) and formats (such as from a WAV file to an MP3 file). A *TransductionModule* instance `subscribes` with a list of attributes specifying the metadata for information that it can convert. When matching captured data is available, the `transduce` function of each

TransductionModule is automatically invoked by the INCA runtime system. The transduced information is then passed on to those *StorageModules*, *AccessModules* or *TransductionModules* that have matching subscriptions for the newly generated data. Additional tagging of metadata to newly transduced data happens in a way similar to that described for the *CaptureModule.* INCA provides a number of *Transducer* services, such as transforming a video file into a series of image frames and vice versa, transcribing handwritten ink, or converting text to speech.

4.2.5 Observing an Application's Run-Time State

INCA includes an *ObserveModule* which developers can use to gain a detailed description of the run-time state of the system. This component contains two mechanisms for obtaining the system's run-time state. First, an application can call this component's `listModules` function to poll for descriptions of all existing modules. Each *ModuleDescriptor* object describes the module type and its state. Additionally, the *ModuleDescriptor* also specifies the subscribed behavior the module supports; for example, the transduction of wav audio into another format. Additionally, the *ObserveModule* also contains callback functions that the infrastructure invokes when the set of existing modules changes or when a module's behavior changes. For example, when a classroom access application changes from subscribing for a particular lecture given by a specific instructor to support a different course and instructor, the *Registry* publishes its change in behavior. The *ObserveModule* can learn about this change in behavior through the `behaviorChanged` callback function; alternatively, it can constantly invoke the `listModules` function to learn of this change. This callback

function enables application developers to build programs that can react to dynamic changes in the run-time environment.

4.2.6 Controlling Specific Application Behaviors

In addition to being able to observe the run-time state, application designers and end-users also may want to control the capture and access of activities. Accordingly, INCA includes a *ControlModule* for modifying the run-time state of a system. Applications can invoke the *ControlModule*'s `control` function to specify the new state (such as stop capturing) or behavior (such as to discontinue the use of timestamp tags).

This component must be used in conjunction with the *ObserveModule*. The `control` function requires a *ModuleDescriptor* parameter. We do not allow a developer to instantiate this object class. Instead, a specific *ModuleDescriptor* handle can be obtained as part of the list of descriptors returned by the *ObserveModule*'s `listModules` function or the `stateChanged` callback function. Internally, each *ModuleDescriptor* contains a unique identifier (UID) private variable that identifies the specific module it describes. We use this UID to associate one *ModuleDescriptor* to a specific module on the off-chance that different modules publish similar behaviors and the infrastructure needs a way to discriminate between them. However, we make this variable and the way INCA uses it to tie a *ModuleDescriptor* and its respective module together transparent to the application developer.

4.3 Summary

In this chapter, we presented an implementation of the conceptual framework

described in Chapter 3, called the INCA Toolkit. The INCA Toolkit consists of three types of reusable building blocks that support specific architectural concerns and encourage developers to decompose the development problem into smaller, more manageable units.

- First, the core building blocks support the capture, access, storage and transduction of information streams.
- Second, within the INCA architecture, we emphasize the role of meta-content in capture and access applications and encourage a data-centric application design model that supports the attribute-based capture, retrieval and management of information. We provide components to facilitate in the attribute-based tagging and querying of information, as well as attribute-triggered garbage collection.
- Finally, the third type of building blocks allows various stakeholders to observe and control the run-time state of a system.

We developed all these components on top of a network abstraction layer that hides details about the underlying communications scheme that connects the different parts of the system. In addition to describing the high-level architectural features embodied by the INCA Toolkit, we presented how the toolkit supports the common functional concerns found in capture and access applications. In this presentation, we discussed various tradeoffs involved with our specific implementation. We provide extremely low-level implementation detail in Appendices at the end of this dissertation.

CHAPTER 5

BUILDING A CAPTURE APPLICATION USING INCA

In this chapter, we demonstrate how the INCA toolkit can be used to develop an audio capture application, known as the Personal Audio Loop (PAL). This application is a near-term audio recording system designed to assist with the resumption of interrupted conversations. In this example, we design the PAL application to run as a standalone application on a single device (laptop or handheld) that users carry. PAL constantly records the surrounding conversation, holding a buffer of 15 minutes of recorded audio. The application automatically discards portions of the captured audio that are older than 15 minutes. Additionally, unlike a tape recorder, this service continues to capture audio even when the user invokes playback of previously recorded information. When the user wants to be reminded of the content of a prior conversation, after perhaps some interruption, she uses a simple interface to jump backwards in the audio stream.

5.1 High Level Architectural Design

At a high architectural level, we divide this capture and access problem into four smaller concerns (see Figure 5-1). First, we design a simple user interface that allows the user to specify a point in the audio stream to review. The user interface requests the captured information and plays the audio to the user. One part of the system must constantly capture and timestamp audio. A third component must temporarily store the information. When any portion of the audio becomes older than 15 minutes, the final portion of the application must discard it.

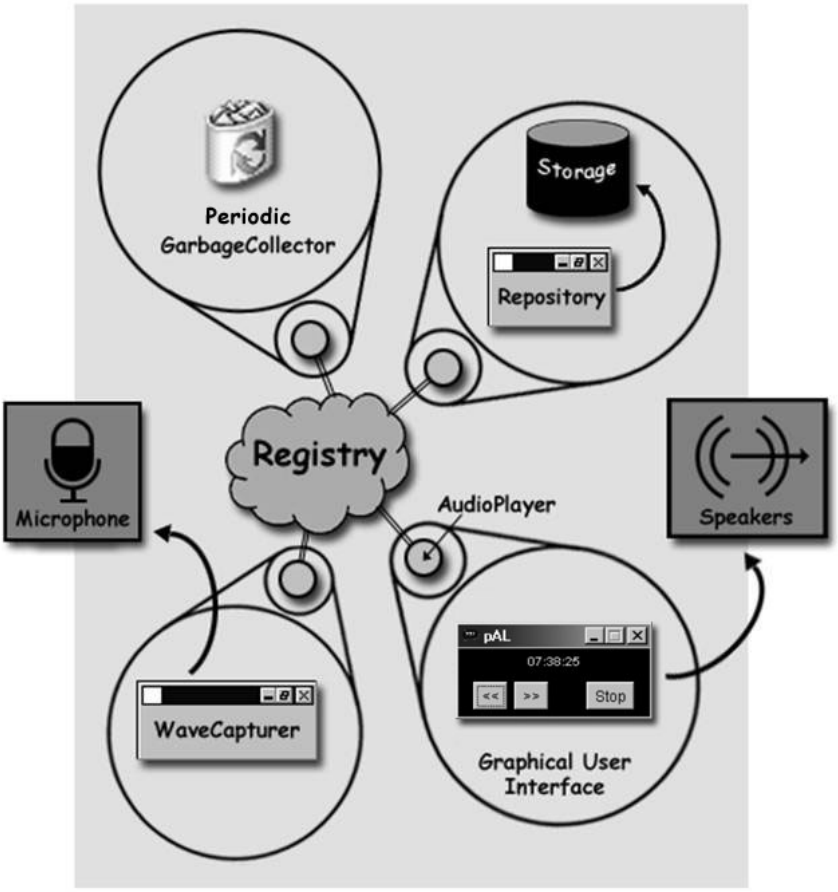


Figure 5-1: The high-level architecture of the Personal Audio Loop application.

5.2 Implementation Details

To begin, we instantiate a *Registry* component in our main program:

```
Registry registry = new Registry();
```

PAL includes a simple GUI for the user to specify the time in seconds, *t*, back from the present, at which audio should begin playback (see Figure 5-2). The user can jump back 30 seconds in the recently recorded audio stream. She can also nudge forward 7 seconds in the event of an overshoot.

Figure 5-2: The simple user interface for the Personal Audio Loop application. Using a model similar to ReplayTV™ (http://replaytv.com), the interface includes a left arrow button that allows the user to cause the application to jump back 30 seconds from its current playback time, a right arrow button to jump forward by 7 seconds, and the third button to end the playback.

We use the predefined *AudioPlayer* component, an extended *AccessModule* in the INCA toolkit, to play the requested audio.

```
// audio player component for requesting audio
// it automatically plays the audio
AudioPlayer audio_player = new AudioPlayer();
```

The application uses this component to issue a query for a minute of audio information starting from the time-point specified by the user. The *AudioPlayer* component then stitches together the audio and sends the content to the speaker for playback.

```
// create a query for information starting from time t until
// (t + 60 seconds)
Query q_start_time = new Query();
q_start_time.greaterThan(new Attribute("TimeStamp",new Long(t).toString()));
Query q_stop_time = new Query();
q_stop_time.lessThan(new Attribute("TimeStamp",
  new Long(t+(1*60*1000)).toString()));
Query q_main = new Query();
q_main.and(q_start_time);
q_main.and(q_stop_time);


// use the audio_player to request the audio (and automatically playback
// the audio)
audio_player.playback(q_main);
```

We use a *WaveCapturer*, an extended *CaptureModule* component in the INCA

toolkit, to capture audio. The *WaveCapturer* component extends the behavior of the *CaptureModule* to support the capture of audio information. In a continuous loop, the *WaveCapturer* component reads audio data available from the microphone input into an array of bytes. Periodically, the `capture` function defined in the *CaptureModule* is invoked, collecting a buffer of audio data and preparing it to be tagged and stored. We register a *Tagger* object to add time attributes that can facilitate the future retrieval of the captured audio. Once initialized, we start the *WaveCapturer*.

```
WaveCapturer wave_capturer = new WaveCapturer("localhost");
wave_capturer.addTagger(new TimeStampTagger());
wave_capturer.startCapture();
```

To store the audio, we simply instantiate a *Repository*. The *Repository* automatically subscribes for all captured information. Additionally, INCA transparently invokes the `retrieve` function in the storage component and returns a list of matching audio chunks.

```
Repository repository = new Repository();
```

To discard audio, we create a special *GarbageCollector* object that periodically discards information. Every minute, this component requests that information older than 15 minutes is disposed. The frequent discarding behavior demonstrated in the *PeriodicGarbageCollector* component can be provided as a reusable component in the toolkit. We intentionally show the full implementation of this behavior to demonstrate how a developer would use one of the low-level INCA modules.

```
import edu.gatech.coc.inca.arch.data.*;
import edu.gatech.coc.inca.arch.format.*;
import edu.gatech.coc.inca.arch.module.*;

/**
 * Class for periodically discarding content older than a fixed amount of time
 *   in this case: 15 minutes.
 */
class PeriodicGarbageCollector
  extends GarbageCollectionModule
  implements Runnable
{
  //============================================================================
  // INSTANCE VARIABLE(S)
  //============================================================================

  //// thread object
  protected Thread thread;

  /**
   * Constructor
   */
  public PeriodicGarbageCollector()
  {
    thread = new Thread(this);
    thread.start();
  } // end of PeriodicGarbageCollector() constructor

  //============================================================================
  // INSTANCE METHOD(S)
  //============================================================================

  /**
   * Run loop for the thread
   */
  public void run()
  {
    while(true)
    {
      try
      {
        Thread.sleep(1000 * 60);
        Query q_time = new Query();
        q_time.lessThan(new Attribute("TimeStamp",
          new Long(System.currentTimeMillis() - (15 * 60* 1000)).toString()));
        gc(q_time);  // remove data older than a certain time from storage
      }
      catch(Exception e)
      {
      }
    }
  } // end of run()

} // end of PeriodicGarbageCollector class
```

5.3 Summary

The complete code source for this application can be found in Appendix D. In this chapter, we demonstrated how the toolkit encourages the application to be decomposed into smaller more manageable components. INCA facilitates the translation of these architectural functions into executable form. In the implementation details section of this chapter, we illustrated the amount of code that we wrote to

support the essential capture and access features of this application.

Previously, Myers and Rosson found that typically 48% of the source code is devoted to the user interface [Myers and Rosson 1992]. In this particular example, well over 67% of the source code we wrote was needed to produce the user interface. However, the access interface is only one of the four core capture and access concerns. Furthermore, we did not need to provide a user interface for the other three concerns. The access interface is thereby the only user interface to this application. Although we needed to devote a significant amount of code to develop this GUI, for this particular application, we achieved the effect we originally desired for this toolkit. INCA allowed the development effort to focus on the essential features of the application. As a result, interaction design and information design became more important than accidental features that are now abstracted.

CHAPTER 6

EVALUATION OF THE INCA TOOLKIT

In Chapter 1, we stated our goals for the INCA Toolkit as:

1. To support the development of capture and access applications. If designers have applications of automated capture and access in mind, INCA can be used to develop that system and it helps focus the design on the essential features of the application.
2. To facilitate the exploration of the capture and access design space. If designers wish to explore new issues or creative ideas, INCA can support these investigations.

In this chapter, we discuss our evaluation of INCA against the above goals. We present the collection of case studies, problem domains studied and applications built by us and by others. These case studies demonstrate that developers can use INCA to create applications that belong within the capture and access design space. Furthermore, some of these applications explore novel capture and access features, such as context-aware video capture or the ability control access to stored information.

We begin by presenting the applications built by developers as part of their research projects. For each case study, we present an overview of the application and motivation from the problem domain, a technical description of the prototype itself, and a description of how INCA influenced the design and development process. The case studies we present are:

- **A large input surface capture & access application:** an application that captures ink written on a large wall surface instrumented with 5 Mimios chained together; the application also provides an interface for reviewing the ink stream.
- **The Walden Monitor:** a mobile capture system that integrates video of the classroom behaviors of children with autism with the researcher's assessments of behavioral variables.
- **Classroom capture applications:** implementations of the original Classroom 2000 / eClass system using INCA.
- **A synchronous discussion tool:** a chat application that records the live conversation between students visiting the same Swiki page and engaged in a synchronous online discussion.
- **A context-aware video capture application:** a system that captures and tags video from a location with context.
- **A token-based access control mechanism:** an approach that associates a set of tokens to each environmental data item captured and stored by the system.

While other developers created the applications described above, we have also used the INCA toolkit to build applications for our own research. We also present the following applications that we designed:

- **Cepher:** a note capture system that supports the capture and retrieval of short notes through a variety of input devices.
- **WebMemex:** an application that logs the user's Web visits and recommends relevant Web pages.

- **eClass v2.0**: our own version of the classroom capture system.

6.1 An Application that Supports the Capture & Access of Writings on a Large Input Surface

Most commercially available electronic whiteboard surfaces are much smaller than traditional whiteboards or chalkboards. For some situations, this size difference alters the way users interact with the boards, prohibiting them from interacting with large amounts of content at one time. In a classroom setting, this constraint can alter the way a lecturer presents and discusses information. As a workaround to this problem, developers often provide extended whiteboard surfaces, such as the one in the Classroom 2000 / eClass project, to display previously captured content. In a design setting, the constraints imposed by a smaller electronic surface can force designers, usually teams of people who are accustomed to writing information along multiple adjacent walls, to alter their practices. In this case, the simple workaround of displaying past information that is no longer editable on separate surfaces will not suffice, because designers need to continue to work with all of the presented information.

As part of a larger research project, a Computer Science Master's student developed a program that chains together 5 Mimio capture devices (http://virtual-ink.com) installed on two walls of a meeting room in the basement of the Aware Home (see Figure 6-1). The project's first goal was to couple this work with the ability for the user to project a computer display onto any portion of these two walls; the Mimios would allow the user to interact directly with the projected display at the walls.

Because people often write on these walls during meetings, the second goal was to allow the users to capture and review notes written on the walls. The student used

INCA to capture the notes written on the board and to provide an interface for reviewing the content. Although this application is fairly simplistic, the student needed to develop it locally, because the software included with the Mimio only supports a single device.

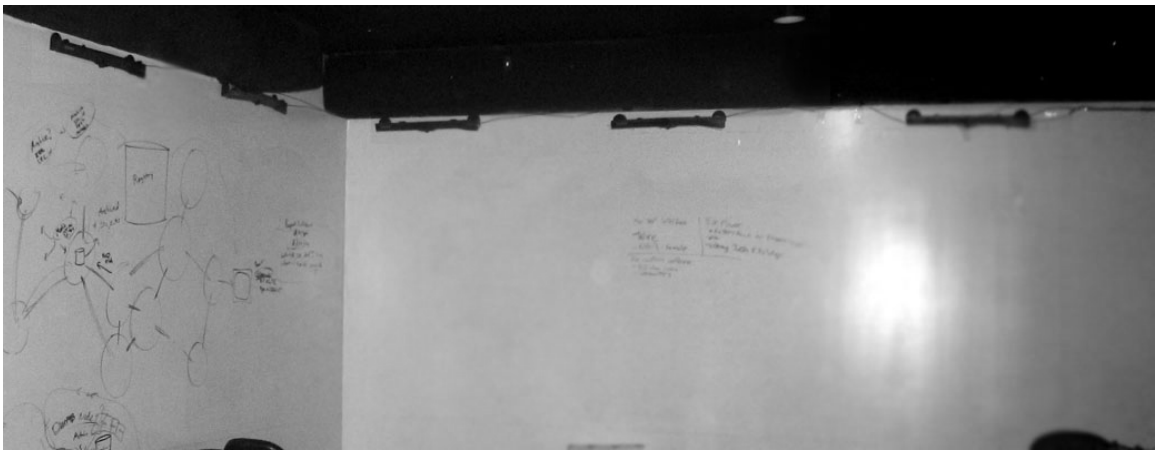


Figure 6-1: Walls instrumented with 5 Mimio sticks chained together.

6.1.1 The Prototype

Before building this particular capture and access application, the student already wrote an application to read continuously pen input events from the 5 Mimio devices. Each Mimio covers one section of the board. As a result, when a user writes a long stroke on the wall that stretches across multiple Mimio regions, the large whiteboard application must read events from each different Mimio device; then it must stitch together the events from these devices and interpret the events as a single ink stroke.

The student used INCA to add capture and access capabilities to the application described above. The capture happens in a seamless manner, in that the very act of

writing on the surface is capturing from the user's point of view. The access interface allowed users to select the start and stop time of the information stream to review. He made the decision to allow the user to specify start and stop time because he wanted to allow them to review more than one meeting at a time when appropriate, because he had observed the practice of continuing meetings at various interrupted times during a day or across multiple days. The access interface includes a large canvas for replay of captured ink strokes and a timeline that allows the user to move quickly through the stream. Additionally, the interface displays thumbnails of periodic snapshots of the board state between the two time points specified by the user. Users can click on these thumbnails to jump to specific points in the captured stream.

6.1.2 How INCA Supported the Development Effort

The student was one of the earliest users of INCA when he used it to develop this application. As a result, the toolkit did not include very many reusable capture and access components that supported specific data streams, such as the *BoardSurface* component to capture ink information. Thus, the student simply used the primitive building blocks to build the application: *CaptureModule*, *Repository*, and *AccessModule*.

Using the basic *CaptureModule*, the application captured ink information and tagged it with timestamp information. The toolkit included a *Repository* at the time and his application used it to store information. The access interface used the *AccessModule* to specify queries for ink information.

The capture and access capabilities of this application were not complex and

required less than one hour to add the capture feature and develop the access interface. The student also reported that the existence of the *Repository* component saved him a significant amount of time because he believed he needed a database back end and did not want to build this functionality himself from scratch. Ultimately, he spent the most time developing the access GUI.

6.2 The Walden Monitor

For his Master's project, one HCI student studied the work practices of teachers and researchers at Walden Early Childhood Center at Emory University. At this center, teachers and researchers help to improve the language and social skills of children with autism (CWAs) by administering early behavioral intervention in the context of typical preschool education activities. Teachers and researchers generate intervention plans and assessments for each child, ages 2-5, through regular observations.

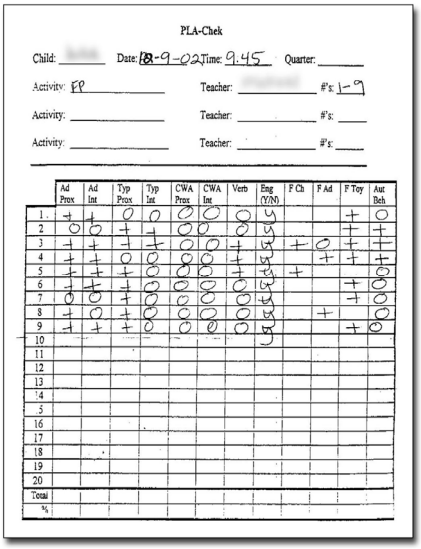
For ten weeks, this student observed classrooms for six hours a week and interviewed key stakeholders (two teachers, two researchers, and three sets of parents). He then designed a mobile capture system that integrates video of the classroom behaviors of CWAs with the researcher's assessments of behavioral variables.

Each calendar quarter, a researcher enters the classroom for ten consecutive days and observes a particular CWA. The researcher captures her observations on a paper spreadsheet (see Figure 6-2) known as a Pla-Chek (pronounced "PLAY-check"), on which these variables are recorded:

- proximity to adult (within three feet)
- adult interacting with CWA

- proximity to typical child
- typical child interacting with CWA
- proximity to another CWA
- other CWA interacting with target CWA
- verbalization (words listed in dictionary)
- engagement
- focus on an adult (if the child is engaged)
- focus on another child (if the child is engaged)
- focus on a toy (if the child is engaged)
- autistic behaviors

The researcher mentally counts a ten-second interval, then records positive or negative results for these twelve variables such as proximity to an adult (within 3 feet) or an adult interacting with the target CWA. The researcher repeats this process twenty times.


PLA-Chek

Child: [redacted] Date: 8-9-02 Time: 9:45 Quarter: _____

Activity: FP Teacher: [redacted] #'s: 1-9

Activity: _____ Teacher: _____ #'s: ____

Activity: _____ Teacher: _____ #'s: ____

	Ad Prox	Ad Int	Typ Prox	Typ Int	CWA Prox	CWA Int	Verb	Eng (Y/N)	F Ch	F Ad	F Toy	Aut Beh
1	+	+	0	0	0	0	0	Y			+	0
2	0	0	+	+	0	0	0	Y			+	+
3	+	+	+	+	0	0	+	Y	+	0	+	+
4	+	+	0	0	0	0	+	Y		+	+	+
5	+	+	+	0	0	0	+	Y	+			0
6	+	+	+	0	0	0	0	Y			+	0
7	0	0	+	0	0	0	0	Y			+	0
8	+	0	+	0	0	0	0	Y		+		0
9	+	+	+	0	0	0	0	Y			+	0
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
Total												
%												

Figure 6-2: The paper Pla-Chek form.

The same data as above are also gathered using the Child Behavior Observation System (CBOS), in which a researcher enters the classroom on a different day with a handheld video camera and records the child for five minutes. Another researcher watches the video and codes the variables on a spreadsheet similar to Pla-Chek with the omission of certain variables: proximity to other CWAs, interactions with other CWAs, and autistic behaviors. These omissions exist because the second researcher may not know which children in videos are CWAs. No visual indicators differentiating the typical children from CWA are allowed at the center. A teacher or other caregiver at the center tabulates the data and includes it in written reports. Parents may see the videos upon request, but they are not routinely shown.

Severe staff limitations require that these tests be administered less frequently than might be desired and never at the same time. The Pla-Chek has two clear disadvantages over CBOS. Because sessions are not videotaped, they cannot be reviewed for accuracy, or be used for demonstrating visually to parents that progress is being made. Pla-Cheks also place cognitive burdens on researchers. They observe children for intervals of ten seconds, which are counted mentally, then record values in a line of cells. Each line is followed by ten more seconds of observation. The next line is filled and the process repeated until twenty intervals are done. Counting time complicates the recording, which requires strict objectivity. CBOS on the other hand requires researchers to devote a large amount of time to code the video; meanwhile, despite this devotion of large amounts of time, the end-result does include fewer variables than the Pla-Chek technique.

6.2.1 The Prototype

The HCI Master's student designed an application to support the individual whose primary task is recording data. Although he initially considered a distributed solution in which cameras mounted in the room collected video and the researcher carried a Tablet PC to record observations, he quickly decided that a localized wearable solution was the most practical and effective approach. The application he designed, known as the Walden Monitor, runs on a Tablet PC and includes a head-mounted bullet camera that is to be worn by the researcher recoding the data (see Figure 6-3).

Figure 6-3: The Walden Monitor prototype runs on a Tablet PC and includes an attached head-mounted camera to be worn by the individual recording data.

Using the Walden Monitor, the researcher observes the child for a ten-second interval and is then prompted by a beep in the earpiece for optimal user awareness and minimal classroom distraction to record behavioral variables on an interface that

mirrored the familiar paper Pla-Chek form (see Figure 6-4). The capture interface used the Quill toolkit [Long 2001] as a gesture recognizer, with a few changes that allowed for the automatic interpretation and tabulation of the observers' data. The data are synchronized to the appropriate intervals in the video, linking all observations about a ten-second interval to the beginning of the video of that interval. Metadata describing when, what, and for which child information is captured is stored in the relational database and associated with the handwritten annotations and video recordings.

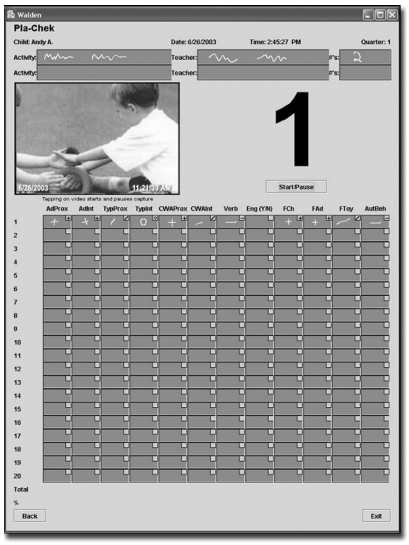


Figure 6-4: The Walden Monitor's capture interface maintained, as much as possible, the look and feel of the original Pla-Chek form.

Two levels of detail are available for access (see Figure 6-5). Users can view a single session (the twenty recorded intervals) using a timeline interface to replay each ten-second video clip next to the observations made for that interval. Alternately, observation columns can be selected to provide more random access through the video

segments. Summary statistics for a session are automatically calculated, and a second view visualizes this summary data across many sessions.

Figure 6-5: The Walden Monitor's access application. The access interface has at the bottom a "macro" timeline that shows an overview of a child's ten quarterly Pla-Chek sessions. The micro timeline at the top right shows the results of the selected session, and the video for that session appears at the top left.

6.2.2 How INCA Supported the Development Effort

After designing the prototype described above, the HCI Master's student delivered the design to a Computer Science Master's student, who developed the Walden Monitor using the INCA Toolkit. Working part-time on this project, this second Master's student used under three month's time to develop the interfaces shown in Figure 6-4 and Figure 6-5. Because the student had no prior experience with Java, nor experience with GUI development, he spent most of the time at the beginning of the project learning the language. After having built the application, he indicated that most of his effort was devoted towards building the GUI. This result meets our original intention of hiding the complexities involved in building capture and access

applications and allowing the developers to focus their attention on designing the interfaces.

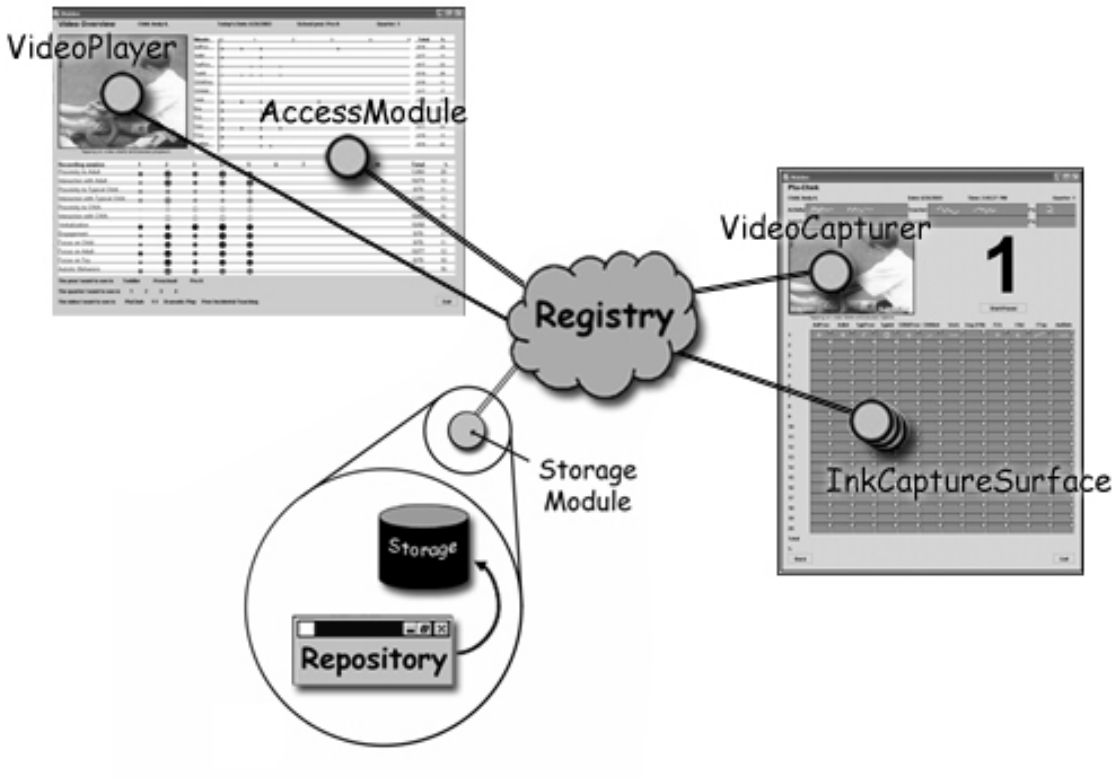


Figure 6-6: The high-level architecture of the Walden Monitor.

The Walden Monitor uses three different kinds of INCA modules: capture modules to record ink annotations and video; a storage module to hold that information for later access; and an access module to provide synchronous access to multiple integrated streams of information gathered from context-based queries. INCA includes two reusable components in its toolkit library that the developer used specifically to capture video data and ink annotations specifying behavioral variables: the

VideoCapturer object and the *InkCaptureSurface* object. When the researcher or teacher writes annotations in regions of the *InkCaptureSurface* object, the application uses the Quill toolkit to recognize the mark made [Long 2001]. The capture interface then tags the ink with metadata describing when and for which child information is being captured, as well as the semantic meaning of the gesture (such as if a behavior is observed, not observed, *etc.*); the application simply tags video with metadata describing when it was captured and which child is captured in the recording.

After each session, the captured information is stored and made available for review by the teacher and/or the parents. The developer used an *AccessModule* to query for all behaviors captured over time, based on their tagged values (if a behavior is observed, not observed, *etc.*). This data is color-coded and then plotted on two timelines to provide a macro view of all the behaviors observed by the teacher and a micro view that shows details of a particular session. Selecting tick marks in the micro timeline causes a *VideoPlayer* component embedded in the access interface to access a video clip of that behavior.

By iteratively designing this application with members of the Emory Autism Center, these students uncovered major usability problems in the capture application. Although this design supported a familiar method of data input, its deployment on a Tablet PC failed. Writing on a tablet is significantly different from writing on paper in two important ways: calibration and resolution. Annotating boxes in the electronic form that were the same size as those on a paper version proved to be noticeably difficult, and the imperfect handwriting recognition resulted in a significant amount of time and effort being spent correcting the data. The research manager also reported that it was

difficult to keep children in the video frame while observing and annotating behaviors.

The original student designer then redesigned the prototype to simplify capture. The new design used screen real estate more economically by replacing the spreadsheet with click boxes for "yes," "no," and "can't tell" (see Figure 6-7). The same set of boxes is used for each recording interval, with the number of the interval noted at the top. He replaced the cells for writing the names of teachers and classroom activities with drop-down menus from which the names can be selected. He also added buttons that can be used to place marks in the timeline when teachers or activities change; these marks remind the researchers to make the changes using the dropdown menus after the session, avoiding interruptions.

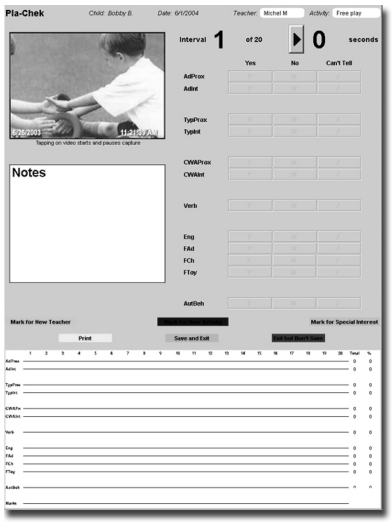


Figure 6-7: The Walden Monitor's final capture interface based on user feedback.

Handwriting and gesture recognition were no longer issues. The application automatically adds data for each ten-second interval to a canvas that renders a quick

review of the CWA's behavior throughout the session. Although graphically, the design changed significantly, the change was isolated to only the capture interface, because INCA encourages separating the capture, access, and storage concerns. Thus, the programmer only needed to replace the *InkSurface* components in the capture interface with buttons that users can push to specify an observed behavior. This event is captured and tagged with the same semantic meaning as the strokes were before, requiring no more changes to the system.

In this case study, not only was INCA able to support development of the application, but the uses informed us about how we might change INCA for other applications. During this case study, we observed a problem that may arise in the capture and access of multimedia information. After capture sessions, the users frequently switch immediately to the access interface to review. At this point, it often appeared to the user that behavioral data was successfully recorded but video was not. In reality, the application did capture the video successfully, but in Java's implementation of video capture, the data resides in memory until the application closes the data source (the camera). At this point, the application begins to save the video. Thus, with that implementation, users had to wait a few minutes before starting the access application for the video to be properly stored and displayed. From this lesson, we recommend that when recording large multimedia streams, applications should capture small segments of video stored immediately, thereby allowing the video to be accessed nearly immediately after recording. Because the Java Virtual Machine has memory constraints, this solution also prevents problems that may arise in continuous capture, when it is not clear if or when the application would ever close the data source.

Additionally, we have reimplemented the *VideoCapturer* object to allow the parent application to store data periodically according to this model.

6.3 Classroom Capture Applications

Not surprisingly, many developers used INCA to implement classroom applications similar to the original Classroom 2000 / eClass system. At a partner university, the Universidade de São Paulo at São Carlos, a Computer Science Master's student created the iClass system (http://iclass.icmc.usp.br) to support the capture of lectures and seminars to generate web-based multimedia documents. At Georgia Tech, the Electrical and Computer Engineering (ECE) Department hired an undergraduate Computer Science student to recreate an application for use in its courses. The ECE Department was similarly motivated in their implementation of the classroom capture system as we were for the original Classroom 2000 / eClass system. The capture and access feature would result in a detailed recording of the classroom lecture. Students can review this information later while they work on projects or study for exams. Because of similarities in the efforts across both universities (with respect to the goals of the projects, the high-level architecture of the end applications, man-months devoted towards the development process), we will only describe the application developed by the undergraduate student at Georgia Tech. In Section 6.4, we will describe a different application built by the same Master's student from the Universidade de São Paulo.

During the Spring of 2003, we interviewed the undergraduate student on a bi-weekly basis while he developed the application. We asked him about his progress, problems he encountered and his design rationales. Although we asked him to record

the amount of time he worked on the project in between the interviews, we must discount the student's self-reports, because they simply matched the amount of time he was paid to work. Additionally, the student just completed his first year of college and had taken no software engineering classes. Thus, not surprisingly, when asked how he divided his time towards the different phases of software development process, we learned that the student spent the majority of the time in the coding phase, testing the code as he went along. He did not consider the high-level design activity to be his responsibility. As a result, we can only provide a *post hoc* report of the experience.

6.3.1 The Prototype

The student never took a course in the original Classroom 2000 / eClass system, which was dismantled at the end of 2001. He was, however, supplied with a high-level description of the system to develop. These high-level features included:

- Capture of the instructor's presentation, her ink annotations on the white board, and audio of the classroom.
- An application that acts as an extended whiteboard to display a recent history of the previous few slides captured.
- Storage of the captured information.
- An application that allows students to review the captured content.

6.3.2 How INCA Supported the Development Effort

Working part-time on this project over the course of approximately three month's time, the undergraduate student used INCA to develop the application described above. By the end of the spring, he had developed the fully functioning

system. We have since adopted many parts of this system in our version of a classroom capture system, which we describe in Section 6.9.

To support the capture of the instructor's writing, he developed a custom application, known as e-Board that is installed on an electronic whiteboard in the classroom. For instructors who teach with prepared presentations, the e-Board application allows users to load a presentation for display on the electronic whiteboard. e-Board is built with a *BoardSurface* component, a specialized *CaptureModule*[2]. *BoardSurface* provides a blank writing surface that listens to pen events and also displays a slide image. As the instructor creates a new blank slide or chooses an existing slide to present, *BoardSurface* captures that slide and automatically tags it with a unique ID (a string containing the name of the machine and the time the slide was loaded or created in the classroom capture and access system). Similarly when an instructor visits a slide, *BoardSurface* captures a visit event tagged with that slide's unique ID. *BoardSurface* also tags ink strokes captured during the slide's presentation with the slide's unique ID. *Tagger* objects, registered by e-Board, automatically associate time, physical location (*e.g.*, classroom number), relevant course information (*e.g.*, class name and instructor name), and relevant system information (*e.g.*, application name and IP address of the machine running the application) to the captured ink strokes, slides, and slide visits. We distributed *BoardSurface* with INCA with the intention that

---

[2] *BoardSurface* is an extension of the *InkCaptureSurface* component we included in the toolkit. The latter component was used in the original capture interface developed for the Walden Monitor application.

developers can use and extend it as described here.

Additionally, INCA includes *WaveCapturer*, a reusable *CaptureModule* specialized for capture of low-bandwidth audio in the .WAV format. The undergraduate student used an instance of *WaveCapturer* to record the classroom. He included in the e-Board application an *ObserveModule* to determine the list of available nearby capture services and a *ControlModule* to start those services. Starting and stopping the audio recording can be coupled with the starting and stopping of the e-Board application.

Originally, the student used the *Repository* object, included in the INCA Toolkit, to store captured information. However, he eventually customized the storage. In particular, he stored relevant information together, such as all the ink information that belonged on a single slide. At the end of the lecture, the storage component processes the captured content, generating images of the captured slides and XML files that describe the captured session and point to the processed content. This was similar to the storage solution in the original eClass / C2000 project.

To allow students to review the captured lecture, he created an interface that retrieves the appropriate XML file. This interface displays the slides and allows the user to initiate playback of audio by clicking on a play button or any ink on the slide (see Figure 6-8). This interface uses an *AccessModule* only to obtain the stored XML file. The student's decision to customize the *Repository* component was somewhat surprising. However, we soon realized that the decision was made in large part because he wanted to optimize the lack of structure in the way INCA stored information.

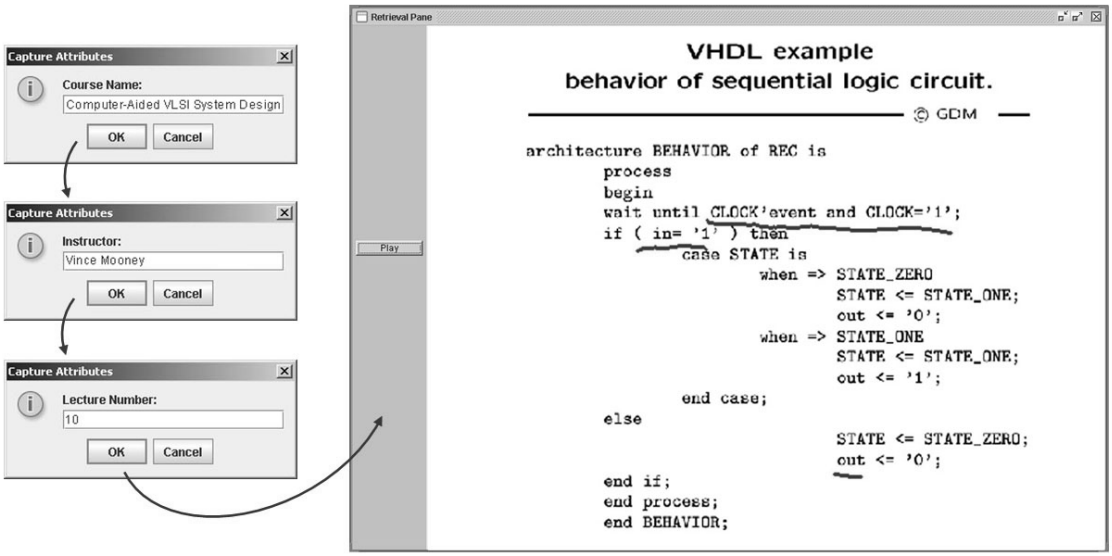


Figure 6-8: Access interface for reviewing captured lectures. A user inputs attributes describing the lecture she wants to review. The access application retrieves the content and allows the user to playback the captured lecture slides and ink annotations synchronized to audio.

As we mentioned previously, a Master's student at the Universidade de São Paulo at São Carlos also implemented a classroom capture application, iClass. That student also developed the system in under three month's time. Interestingly he also made the same architectural design decisions. Again, the student opted to customize the storage component. He developed a database schema to specifically support the capture of classroom materials annotated with *who*, *what*, *when*, *where* metadata. His application also generated XML documents that contained the *who*, *what*, *when*, and *where* information describing the captured session and provided queries necessary for retrieving specific content. The access interface was created using Java Server Pages that would then use the *AccessModule* to query for information as specified in the XML.

6.4 eMeeting: A Synchronous Discussion Tool

The Intermedia research group at the Universidade de São Paulo São Paulo at São Carlos is primarily interested in multimedia and hypermedia research, but has used capture and access as a means of simplifying the authoring of multimedia documents. In their research, they have adopted the use of CoWebs as asynchronous discussion spaces [Guzdial *et al.* 2000]. The CoWeb allows users to easily coauthor Web pages. When an instructor creates a CoWeb for her class, students often use it as a way to engage in asynchronous discussions with each other, as well as with the instructor. In particular, the Intermedia research group has investigated how previously captured content can be turned into anchors for new asynchronous CoWeb discussions. This concept was first introduced in an effort at Georgia Tech to link the CoWeb to the eClass system [Pimentel *et al.* 2001].

To extend this idea, the same Master's student who developed the iClass application discussed briefly in the previous section also used INCA to develop a chat tool, known as eMeeting. This tool supports synchronous discussions between students, initiated when they review the lecture or when they view CoWeb pages that contain discussions about specific points within a lecture. The eMeeting application can add the transcript of the synchronous discussion back to the CoWeb. Whereas a typical discussion on the CoWeb may involve some lag time between posts, eMeeting allows students to engage in a quick and instant conversation about a lecture topic if they are online at the same time.

6.4.1 The Prototype

eMeeting provides features similar to a simplified instant messaging application, while also providing a whiteboard space. The synchronous communication provided in this particular chat tool can be viewed as a capture and access problem. The tool captures text and ink information, and then quickly makes it available to any clients interested in this content. The application can also store the captured content and adds it to the anchoring CoWeb space. Any number of users can participate in the discussion; and multiple discussions can occur at the same time.

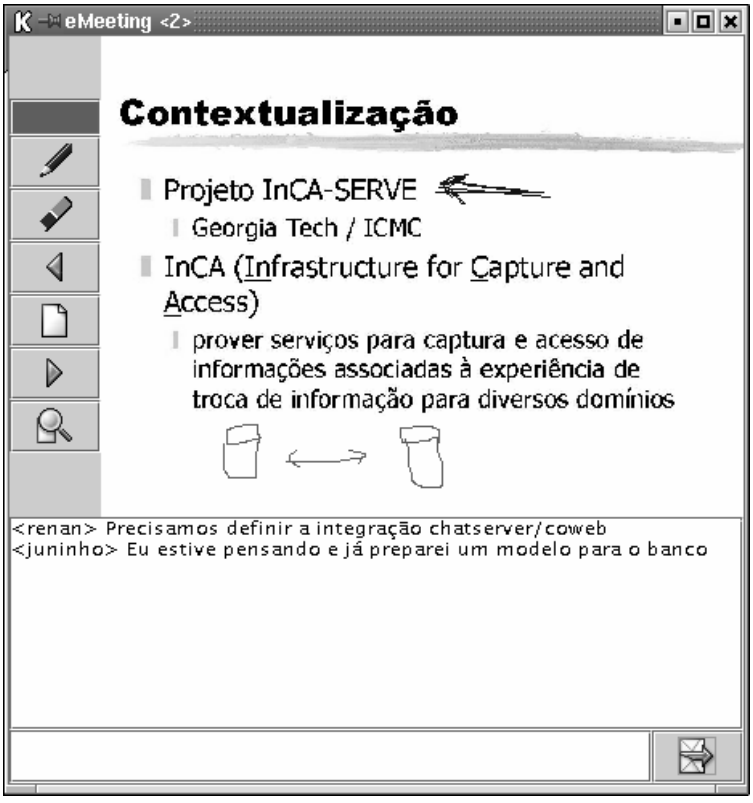


Figure 6-9: Chat application allowing people to share text and ink messages. (Picture courtesy of Maria Pimentel)

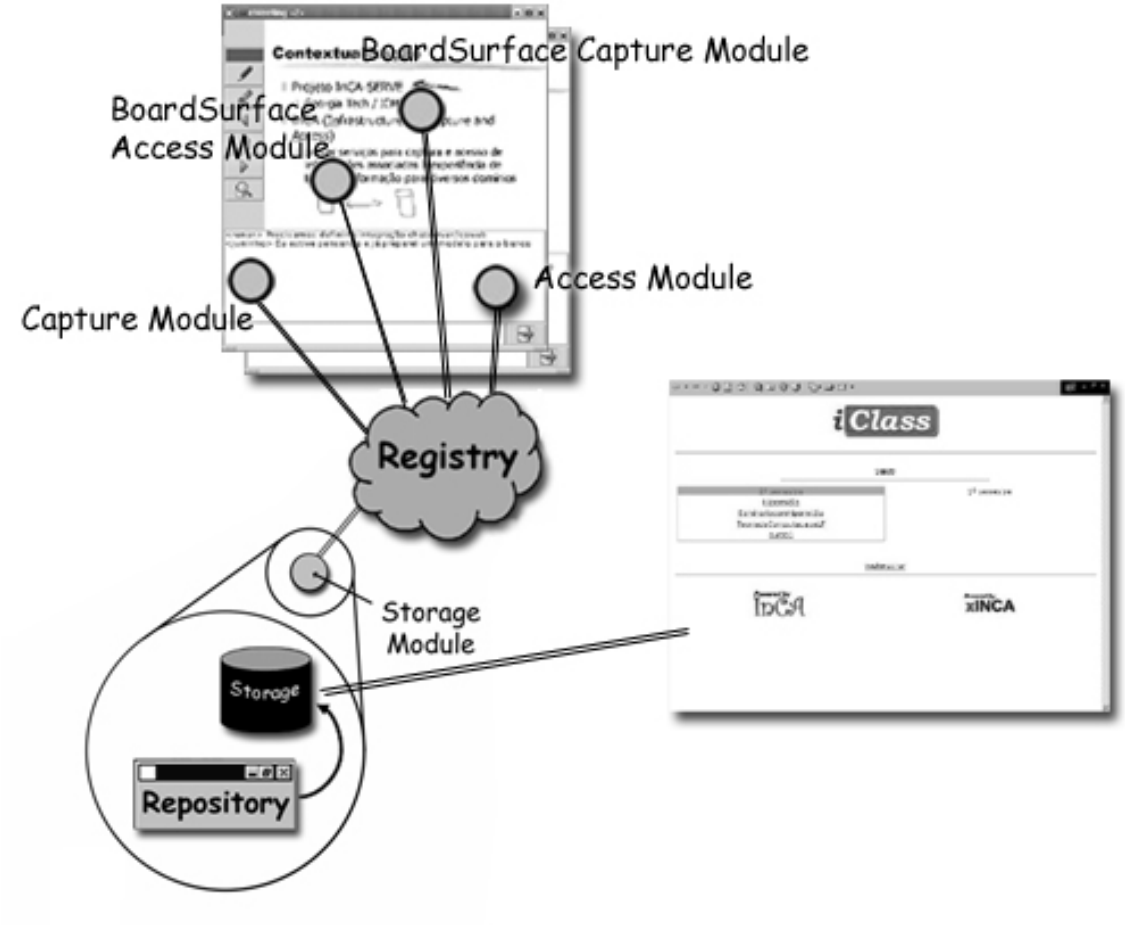


Figure 6-10: The high-level architecture of the eMeeting application.

6.4.2 How INCA Supported the Development Effort

The student developed eMeeting using a *CaptureModule* to record the text messages the user types and sends and an access module to subscribe for text messages generated by other users. At the time when the Master's student developed this application, the INCA toolkit included a whiteboard component that supported the capture presentation slides and ink information and a second whiteboard component to retrieve and display the information. As a result, the student needed only to create a

custom whiteboard component that supported both the capture and the access of information.

Each CoWeb page has its own URL. The eMeeting application uses this URL as a tag for the captured information; in doing so, the application can easily subscribe for content captured and tagged with this identifier. INCA immediately redistributes this content to all access modules, which then display the information. Another advantage to tagging information with the CoWeb URL is that stored content tagged with this URL can be retrieved and added to the CoWeb page. Thus, students will see all of the new information when they use that particular URL at a later time.

eMeeting's use of INCA demonstrates that after a student has used the toolkit once, he is likely to continue using it with subsequent projects. Furthermore, this particular project is interesting because it explored capture with two kinds of access: an immediate access that allowed students to see each other's text messages instantly and provided instant messaging like capabilities, and an eventual access that tied the conversation transcript back to the CoWeb.

6.5 A Context-Aware Video Capture Application

Many people capture home videos of their family members to remind them of special occasions or even just of everyday life. These videos accumulate over time creating a mass of media that is difficult to catalogue or to revisit. As digital video becomes more pervasive, opportunities arise for supporting more flexible means of capturing, cataloguing and reviewing family videos. An access application, such as the Family Video Archive [Abowd *et al.* 2003], could deliver video sequences that matched

queries of the sort, “Show me all of the videos from the 1960’s with my Grandma in them,” or “Let me see all of the important events in my life.” However, this application requires that videos are tagged with context such as location, time, and people present within the field of view and possibly even higher level concepts, such as significance or importance to particular individuals.

To alleviate the need for users to manually tag their videos, one might imagine a video capture application that is aware of the current context of a situation and adds that information to the captured video. Thus, context-aware video capture is compelling for two specific reasons. First, adding context information to captured video can help with its storage and retrieval. Secondly, it can result in more efficient capture of information. For example, the user can specify that a camera should only record when a person is in a location. As part of a class project, two Computer Science Master’s students used INCA to develop a context-aware video capture application that automatically captures and tags videos with context information. The work assumes an environment, such as the Aware Home, that is instrumented with cameras and location-sensing capabilities. The students never used INCA prior to this class and they were not required to use INCA in this class. However, they decided to build their application using INCA for two reasons. First, they believed it would facilitate the capture and access of video data. Secondly, they believed that INCA would provide a useful intermediary between the application and the Context Toolkit [Dey 2000].

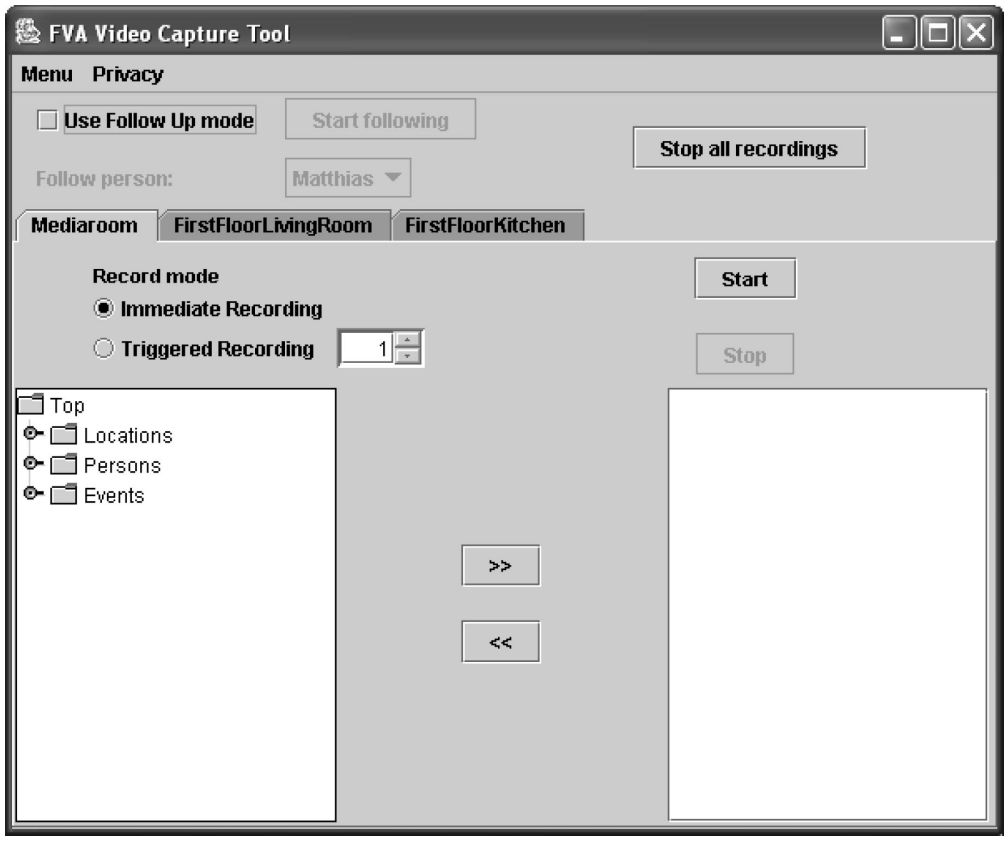


Figure 6-11: Tool for specifying the capture behavior of cameras in a location aware environment. (Picture courtesy of Matthias Gauger & Andreas Lachenmann)

6.5.1 The Prototype

The two Master's students developed the capture system using both the INCA and the Context Toolkits. From the Context Toolkit, they obtained a list of all the people present in the different rooms of the house. Their application uses this information to tag video captured from each location. Typically tags are generated by the self contained application. This application demonstrates how the application can obtain metadata from a context infrastructure, using the *PeoplePresentTagger* object we included in the toolkit. The application can also use this information to trigger the capture of video in specific locations. The students provided an interface that allows the

user to control the recording of several cameras at one time (see Figure 6-11). The user can choose the recording mode, start and stop the recording of a single camera or the whole system, add additional tags by hand to the video currently captured, and define the list of people for whom their privacy should be preserved.

For each camera in the system, the interface provides a tab that allows the user to modify the capture settings for that device. The user can instruct the device always to record and to tag the captured videos. Or, she can instruct the device to record only when certain people are in that space. The user can define the number of people that need to be present in the room before the camera begins to record. The developers included this feature to prevent the camera from capturing scenes where only one person is present and therefore probably nothing worth recording is happening; but rather, the recording should only happen when a person has the opportunity to interact with someone else. Although one can imagine situations in which this approach is naïve, it was reasonable for the case of focusing on the capture of digital family videos. Finally, all cameras also can be instructed to follow a person.

The students did not build an access application. Instead, they developed this application to capture videos that could be accessed through the Family Video Archive application, a system they developed prior to taking this course, in which they wanted to explore this particular capture feature.

6.5.2 How INCA Supported the Development Effort

INCA included a *Camera* component the students used to record video. Each camera registers a *PeoplePresentTagger* object that uses a Widget from the Context

Toolkit to obtain a list of people present at the location where the camera is located. This *Tagger* object tags captured video produced by the *Camera* with this list of names. The *Camera* also adds the time and location information to the video.

Additionally, the tool developed by the students also uses a Widget from the Context Toolkit to subscribe for changes in the people present information of every location where cameras are located. The tool uses the acquired context to turn on and off the recording behavior of different cameras. To turn on and off the recording of the different cameras, the tool uses the *ObserveModule* to get a list of the available devices and uses the *ControlModule* to change the active state of that *Camera*.

The students designed and developed a working version of the capture application in four months, working part-time. When they tested the application, however, they encountered an important problem: the video incorrectly captured scene boundaries. The video did not begin to record until seconds after a person enters a location. Similarly, the video did not stop recording until seconds after when a person leaves a location. The videos recorded in initial testing, thus contained significant gaps between a person leaving one room and entering another. For the specific camera hardware they used, the Java Media Framework takes a few seconds to initialize the camera before it can actually record video. We redeveloped the *Camera* component in a manner that does not rely on JMF to actually start and stop the recording. In the new implementation, the *Camera* component reads video frames from the device when it is in record mode and stops reading frames when it is not; the component then reconstructs the frames into a video stream.

The students also encountered problems with the location system sometimes not

detecting when a person enters or leaves a room. Even when it functions correctly, the location system takes time to read RF ID tags worn by people entering and leaving the room; additionally, the RF ID tags need to be interpreted into names of people. Collectively, the delay between when a person enters or exits a space and when the location system returns a list of people present to the cameras and controlling application creates a situation in which scene boundaries will not exactly align with reality. This problem possibly indicates the need to buffer content, in a manner similar to StartleCam's use of in memory video data [Healey and Picard 1998], so that delays in obtaining sensor data does not result in loss of captured data.

6.6 A Token-Based Access Control Mechanism

Capture and access applications need to secure collected data for obvious privacy reasons. In his privacy and security research work, a Computer Science Ph.D. student explored associating a set of tokens to each environmental data item stored within the capture and access system [Iachello and Abowd 2005]. Token-based access control differs from traditional techniques that employ access control lists in that the user, not the system, keeps track of tokens.

6.6.1 The Prototype & How INCA Supported the Development Effort

This researcher was able to add quickly a token access control mechanism to INCA. First, he created the Token Manager continuously communicates with the *Registry* and provides it a set of tokens. When an application invokes the *CaptureModule's* `capture` function, the *CaptureModule* sends the captured *DataObject* to the *Registry*, which would then deliver the *DataObject* to those access,

storage and transduction components interested in the content. He modified the *Registry* component to associate *DataObjects* with tokens before passing them to the *Repository* for storage; additionally, he added the ability to verify that an *AccessModule* has provided with a *Query* a list of tokens that matches the tokens stored with the requested *DataObjects*. He modified the *AccessModule* to provide tokens when making a request. Finally, he modified the *Repository* component to now store not only the *DataObjects* and their attributes but also a list of associated tokens.

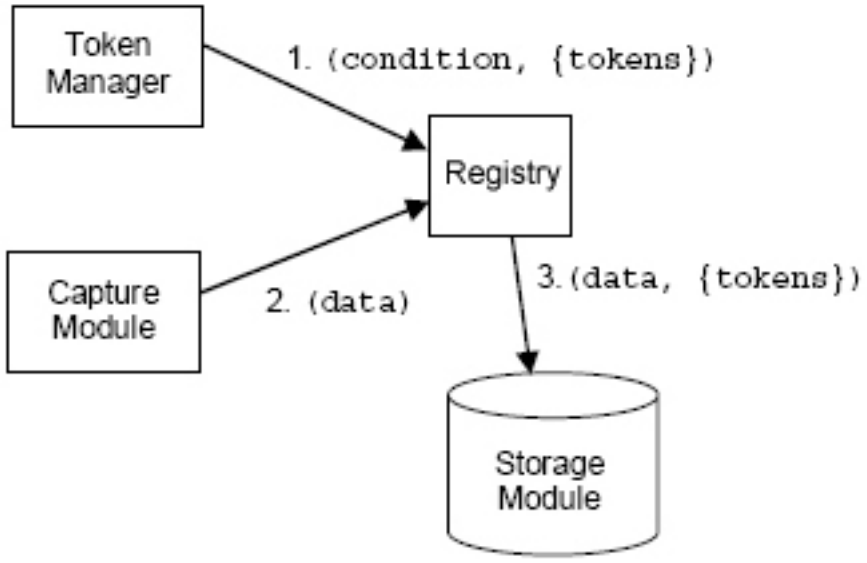


Figure 6-12: The token-based access control mechanism during the capture phase. (Picture courtesy of Giovanni Iachello)

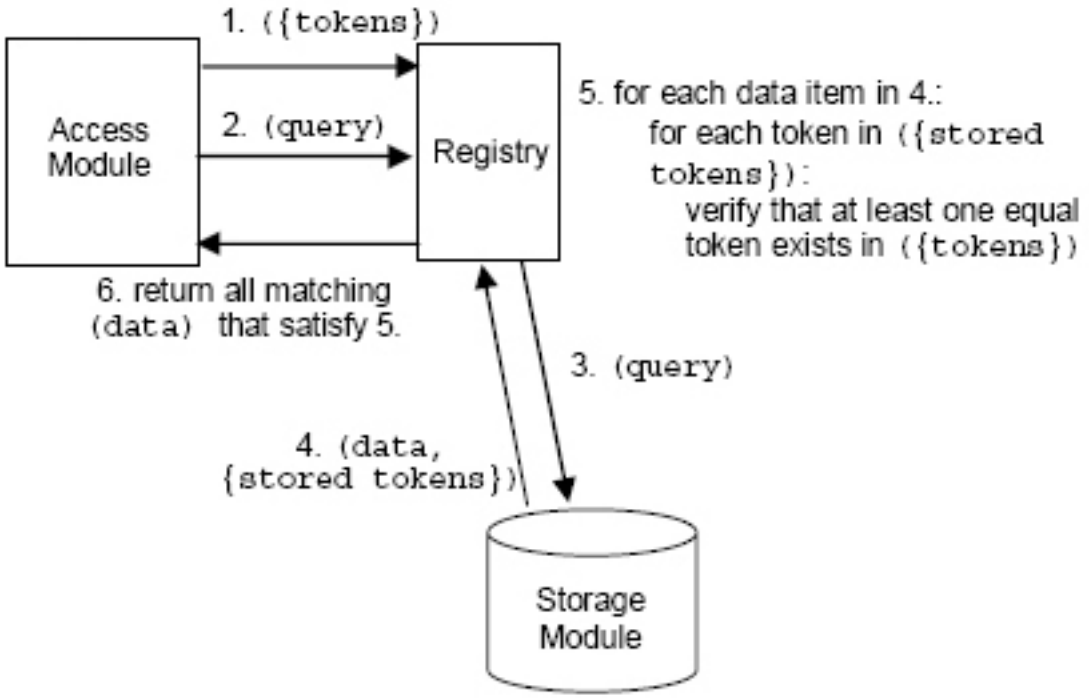


Figure 6-13: The token-based access control mechanism during the access phase. (Picture courtesy of Giovanni Iachello)

6.7 Cepher: Application for Capturing Short Important Thoughts

Cepher (a Hebrew word meaning "writings" and pronounced "safer"), is a prototype system designed to handle the capture and access of short notes to augment a person's ability to record and later use small pieces of potentially interrelated information. Some examples of short important thoughts that users might record are "to do" items, groceries, potential thesis topics, and contacts. These notes all have certain characteristics in common: they tend to be temporary, viewable, mobile, postable, transferable, small, light, short, and easy to create. This form of note-taking differs from note-taking in structured settings [Lin *et al.* 2004]. Improving on current practices will

require physical and digital artifacts, flexibility, multi-modality and ubiquity [Hayes *et al.* 2003]. People would often like to use a variety of methods to immediately capture these thoughts instead of waiting until later for an available recording mechanism. People also require flexible access from a variety of viewers. Thus, the special needs of informal note-taking can best be met using a confederation of devices.

A ubiquitous system for recording and accessing notes, ideas, and other information could help off-load users by augmenting their memories with contextually encoded easy to access digital data. Cepher integrates interesting wireless technologies, small server capabilities, context collection through a variety of devices, and a usable interface for quick note-taking to support users in capturing and accessing the notable details of daily life.

6.7.1 The Prototype

Cepher is a traditional client/server application for mobile users and their array of personal information devices. The system uses the Intel Personal Server [Want *et al.* 2002] as the hardware platform for the server portion of the client/server model. Traditional mobile devices access the data storage and services of Cepher through client applications designed for individual platforms. Users record personal information in whatever modality is most appropriate for their current situation and available capture device; we supported the capture of typed text and handwritten notes, with the intention of later adding audio notes. Users can access this data, stored on the Personal Server, in whatever modality is appropriate for their current situation and available access device. Because all data transfer occurs over Bluetooth, users can successfully capture and

access their information through services in the environment or other mobile devices without ever needing to physically interact with the Personal Server and without any network capabilities inherent to the environment itself.

6.7.2 How INCA Supported the Development Effort

We originally designed the system such that capture and access applications would simply use the *CaptureModule* and *AccessModule* objects provided by INCA. Additionally, we simply launch the *Repository* component on the Personal Server to store captured information (see Figure 6-14).

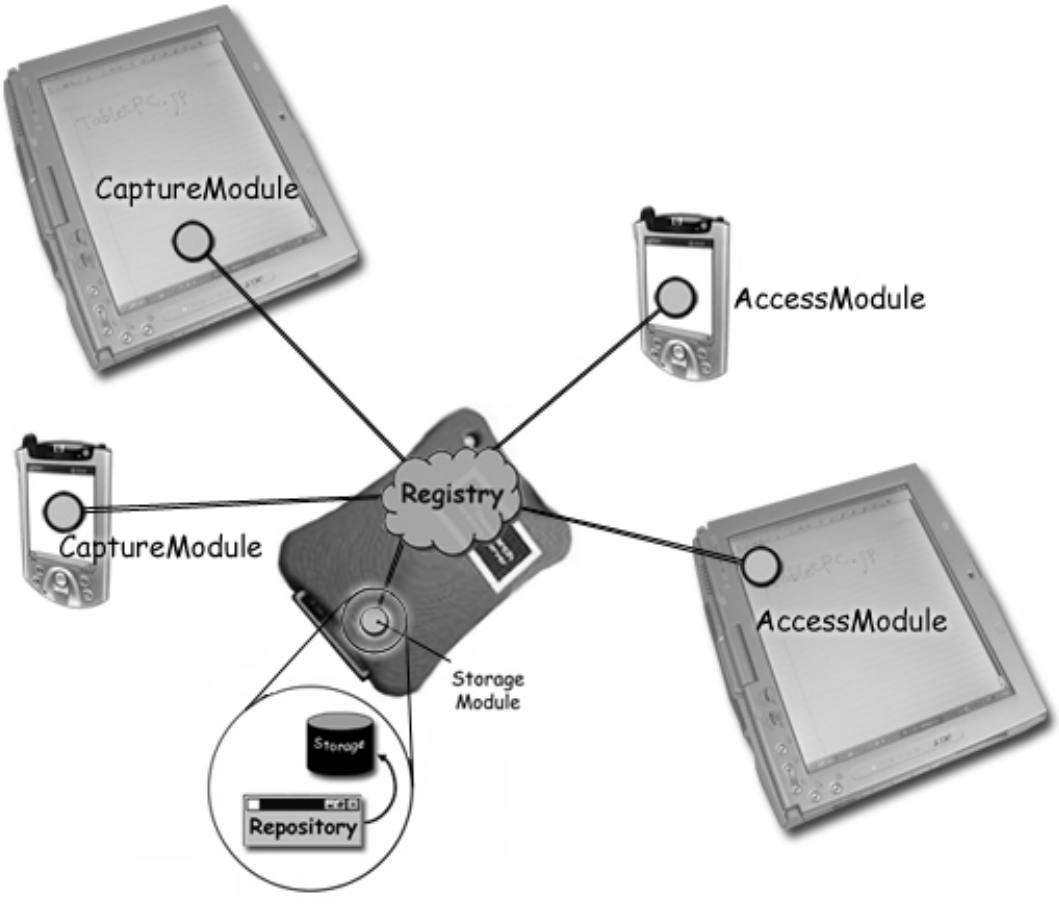


Figure 6-14: The high-level architecture of the Cepher application.

However, we quickly discovered several problems. To retrieve information from the Personal Server in an acceptable amount of time, a database back-end was required. At the time, we had two types of *Repository* objects; the *Repository* can use a hierarchical file structure or a MySQL database. MySQL has a large footprint and consumes much of the processing cycles on the Personal Server. As a result, we extended the *Repository* object to use pure Java databases, PointBase and HSQLDB. Although, the databases take a short amount of time to load table files at start up, they performed comparably to MySQL, as we presented in Chapter 4. They offer as an advantage a very small footprint. We also encountered a second problem that pertained to network and power consumption. INCA requires constant network connections between the capture and access clients with the *Registry*, which would run on the Personal Server also. The constant connections consume lots of resources, most significantly network bandwidth and power.

Given the problems described thus far, we decided to address them through the creation of a different version, developed specifically to run on the Personal Server known as pINCA (short for Personal INCA). Because the *Registry* and *Repository* components would both be physically located on the Personal Server, we coupled the two components and removed the network communication that tied the two together. Additionally, we opted to use the PointBase database as the back-end storage component used by the *Repository* object. We modified the way capture and access components communicated with the *Registry*. Instead of maintaining a constant network connection, a module connects with the *Registry* when a new message needs to be communicated between them. Additionally, we repackaged pINCA such that the

version placed on the Personal Server includes only the minimal number of required Java classes. These steps resulted in a platform specific version of the toolkit, in particular one that runs more efficiently on the Personal Server than the original version.

6.8 WebMemex: Capturing & Remembering Web experiences

History mechanisms in standard Web browsers are impoverished, requiring users to recall hard-to-remember features of the Web page, such as its URL or title. As people experience the Web, they visit many sites and view many documents; however, when people want to retrieve previously seen information, too often they have either forgotten to bookmark the relevant URLs or the local history mechanism of the browser machine they used some time ago is not accessible or not adequate for the current retrieval task.

Inspired by the value of Web capture in eClass lectures, we wanted to build a more general capture service in support of several access services: the suggestion of related URLs that a user has seen in the past (in the spirit of the Remembrance Agent [Rhodes and Starner 1996]); an explicit search over a user's complete Web history; and a collaborative filter to suggest relevant Web sites to friends (see [Macedo *et al.* 2003]).

6.8.1 The Prototype

The WebMemex service is provided through an augmented Web proxy server, resolving the problem that existing history mechanisms are local to a single machine. Standard Web browsers can be quickly configured to talk to an HTTP proxy. When a user surfs the Web, her browser will request Web pages from the proxy server. The

Web proxy server retrieves the request and serves it to the user on the requesting client browser. If the user has capture and access services enabled, the Web proxy server will react appropriately when the document being returned is of the content type text/html.

If the user wants her session captured, Web visits are tagged with the URL, the title, up to 10 keywords for that Web page, the time that Web page was visited, the IP address of the browser machine, and the user's ID. The captured information is stored in a property-based *Repository* until it is later accessed. Two separate access services are supported implementing different integration techniques. To support the recommendation of related URLs to what the user is currently viewing, WebMemex queries for Web pages matching keywords and displays them in a small popup window (see Figure 6-15). To support the search over a user's Web history, information is retrieved based on queries explicitly set by the user. Once found, users can also review the trails surrounding a particular Web page visit derived from a stream of Web visits and then temporally integrated.

6.8.2 How INCA Supported the Development Effort

To facilitate a content- and/or context-based history search mechanism, we used the *WebMemex* component to capture an annotated Web history. We registered a number of existing and custom *Tagger* objects to help associate the user's ID, time and location to each captured Web visit (in addition to the keywords). The access interface consists of custom Java Server Pages that use an *AccessModule* to handle search queries for previously captured Web visits.

This annotated Web history also enables a number of other access features, such

as the automatic recommendation capability. As a user browses new Web pages, a different access application suggests related URLs that the user has visited from the past. An *AccessModule* requests the last Web visit handled by the proxy server (which is currently viewed by the client browser). By taking the keywords of this Web page, the access application can query for previous URLs that she visited with matching keywords. This information is displayed in a pop-up window.

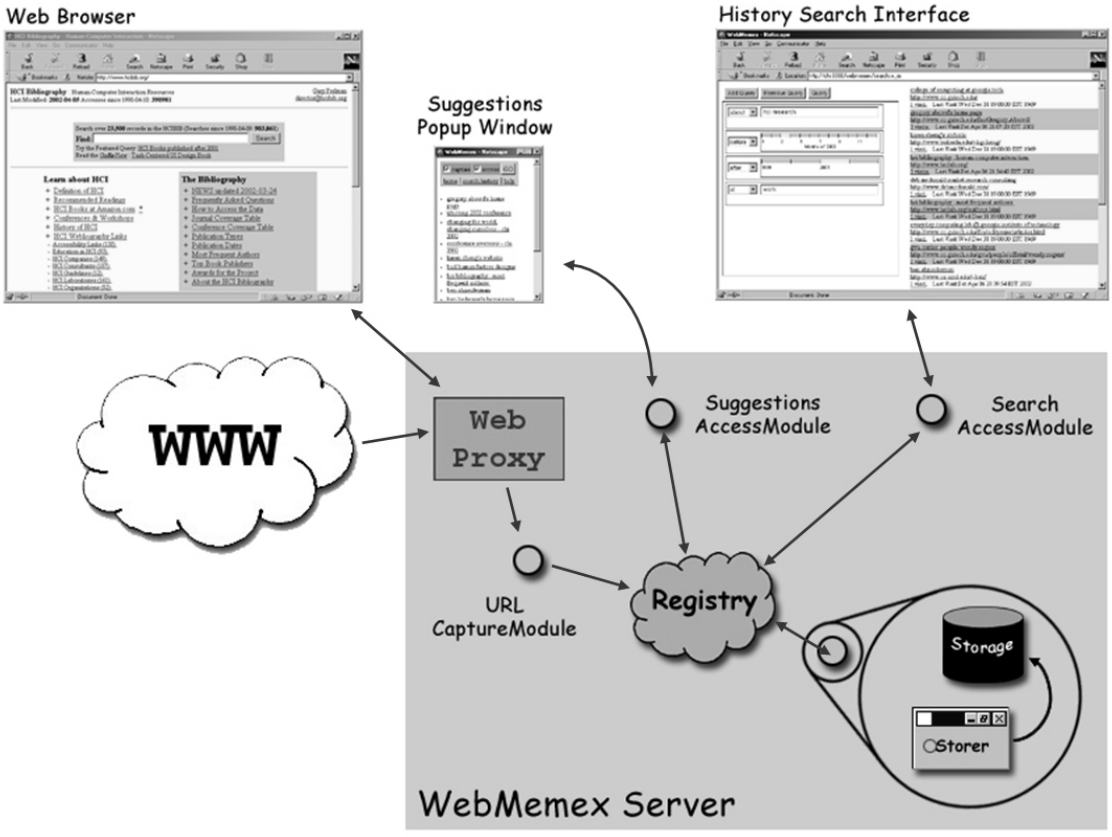


Figure 6-15: The high-level architecture of the WebMemex application. WebMemex is an enhanced Web proxy server to capture Web history and provide access services.

Although we originally developed WebMemex to support individual users as they surf the Web, with colleagues from the Universidade de São Paulo at São Carlos, we have since extended this service to investigate the sharing of Web histories within a social network. We developed a component that communicates with Yahoo's Messenger service to authenticate the user's login and obtain her list of friends, which we considered the user's social network. We were able to replace the less sophisticated user authentication component easily because this modification happened as an isolated change from the rest of the capture and access application. We then modified the access behavior to allow information sharing between users if they exist in each other's buddy list. Thus instead of querying for information tagged with only that user's name, the application also requests information tagged with her buddy's names as well. These changes greatly altered the model and use of WebMemex while requiring very little modification to the code, in large part due to the abstractions made possible through INCA.

6.9 eClass Revisited

We created eClass v2.0 for two specific purposes. First, we wanted to use INCA to capture a course on Principles of User-Interface Software in Fall 2003, and the previous version of Classroom 2000 had long since been dismantled. Second, we wanted to explore including features to the capture system that did not exist in the original Classroom 2000 / eClass system. Beyond our own additions, we describe a short project assigned to students in this course. We used this assignment to study the ability of others to build applications on top of an existing application developed using

INCA.

Because others had created versions of the classroom capture system, as described in Section 6.3, we re-used parts of their system where appropriate. However, we used our original *Repository* component instead of the custom versions that had been developed because flexibility was considered to be more important than the efficiency that might have been afforded with use of a custom repository. We created an HTML based access interface to replace both the original access interface and the one subsequently developed using Flash. Finally, to make the interface match the original system, we added to the ability to capture Web visits for those courses in which the instructor displayed additional materials on the Web with the traditional Power Point lecture slides and whiteboard annotations.

6.9.1 The Prototype

The access interface runs as a standard HTML-based Web interface allowing users the freedom to access classroom data from any web-enabled machine. The interface, composed of custom built Java Server Pages (JSP), instantiates an *AccessModule* to request captured information tagged with the course name and date specified by the student reviewing the lecture. It then presents the lecture as a sequence of discrete slides in the same order used by the instructor during the lecture, regardless of what ordering might have existed in the prepared slides. Users can examine a timeline that indicates the important events taking place during the lecture, such as a slide being created or visited or a Web page being viewed. For further details, a user may click any portion of the timeline or the handwritten ink to begin playback of the

audio corresponding to that portion of the lecture. In this case, the *AudioPlayer*, a toolkit component for playing back audio (using an *AccessModule*) requests audio chunks and begins playback until the user clicks the stop button.

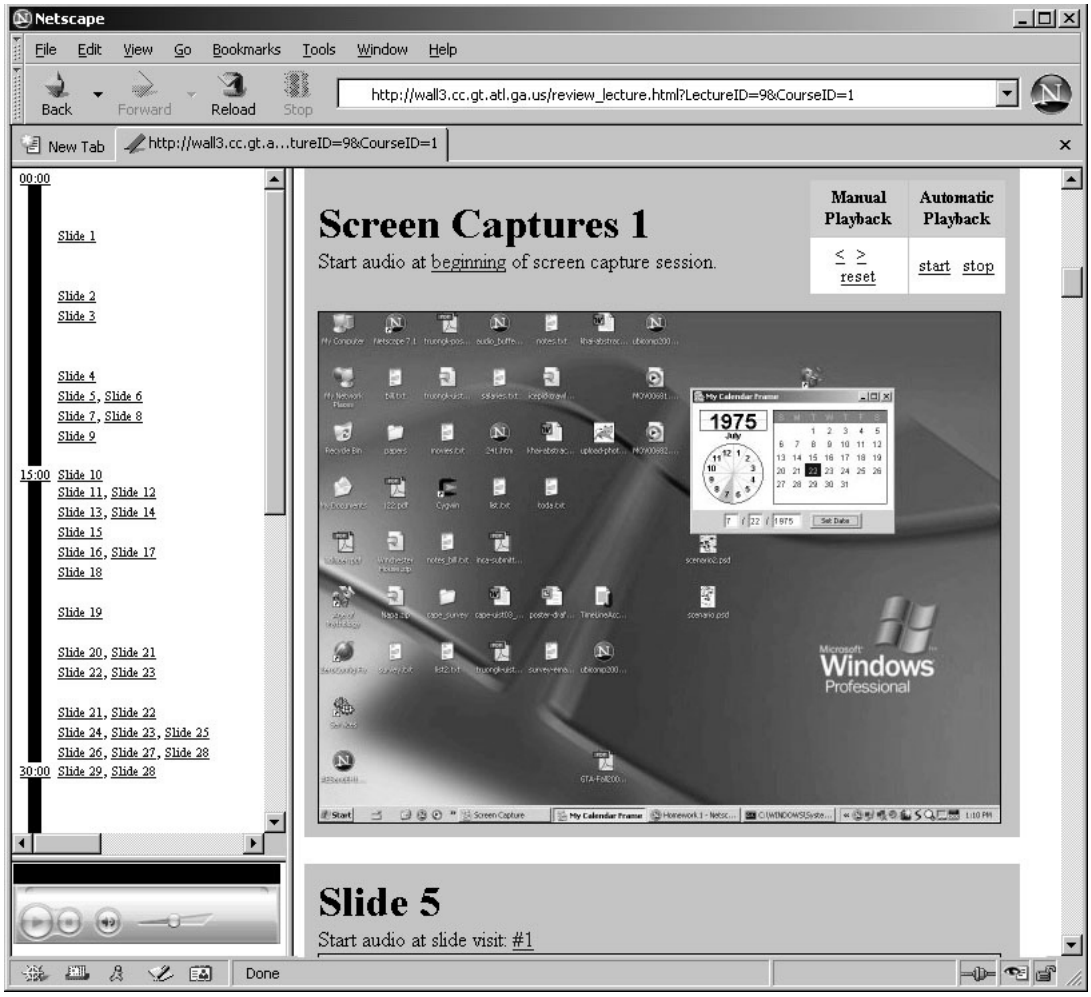


Figure 6-16: Access interface with ability to replay screen captures.

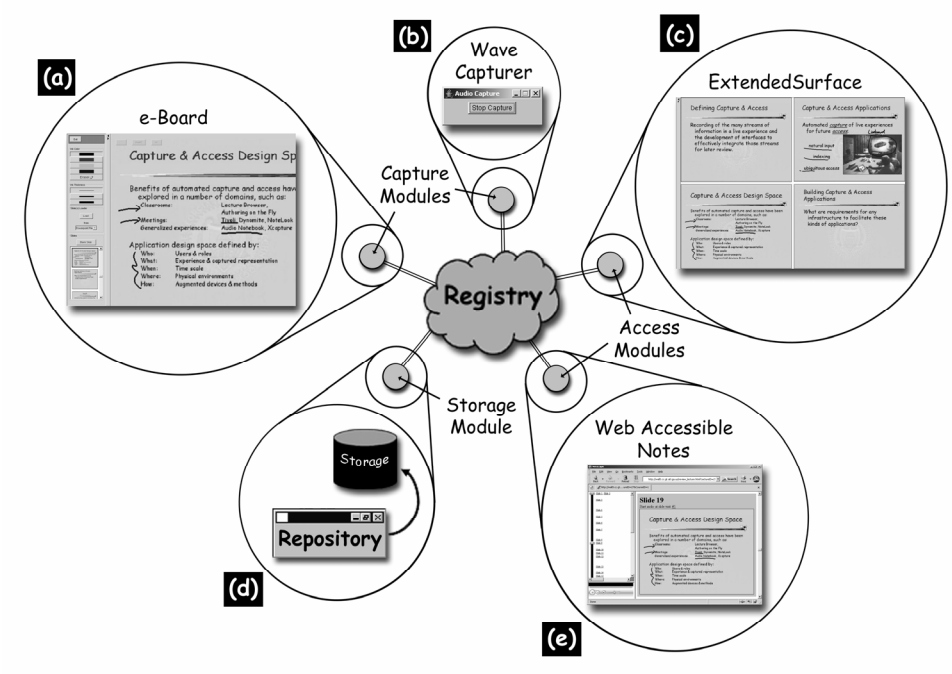


Figure 6-17: The high-level architecture of the eClass system. (a) The e-Board application built using the *BoardSurface* capture module provides a writable presentation surface that allows the instructor to present prepared slides and/or write on a blank surface. (b) A separate component built using the *WaveCapturer* capture module records audio during the lecture. (c) The ExtendedSurface application shows a history of slides captured during the lecture. (d) A Web access interface that includes an audio player, a timeline of the captured lecture, and the slides and their annotations automatically generate using JSP allows students to review the captured lecture. (e) A generic *Repository* object stores all the captured data.

We used the *WebMemex* component described in Section 6.8, a specialized *CaptureModule* provided by INCA, to record Web pages requested by a client browser. Acting as a Web proxy, it handles the HTTP requests and logs pages visited. INCA provides this Web proxy application as a general capture service. In eClass, *WebMemex* helps capture pages viewed in class.

6.9.2 Our Extensions to the System

After recreating a version of the original system, we experimented with adding additional capabilities as well to understand how difficult it would be to add these once

the original system had been built. Not all instructors use electronically prepared presentation slides. As a result, the hurdle for using eClass can be lowered with support for scanning materials on which an instructor might make annotations and for the use of blank slides on which an instructor can write, similar to a traditional whiteboard. This feature is also desirable at times during class when illustrations or examples are written on paper and needs to be integrated with the rest of the captured electronic lecture notes. We developed a custom application, *eScanner* to run on the computer connected to a scanner. This application continuously scans material until a blank image is detected. As images are scanned, they are captured and made available to any application interested in them. *eScanner* publishes that it is a scanner application located in a particular room. The e-Board application was extended to use an *AccessModule* to subscribe for slides created in the classroom and to include a GUI button that activates the loading of slides from the scanner. The e-Board application uses an *ObserveModule* and *ControlModule* pair to request the scanning of any material the instructor has placed on the scanner.

Instructors also occasionally show demonstrations in class. Thus, we also created an application that captures screenshots of a PC in the room during a demonstration. The screenshots can be later replayed in succession to allow students to review the demonstration (see Figure 6-16). We eventually extended this feature to allow the instructor to grab screenshots from any PC installed in the classroom. For example, while an instructor demonstrates how to code an application, he can grab a screenshot of the code, and then annotate it on the e-Board application (see Figure 6-18).

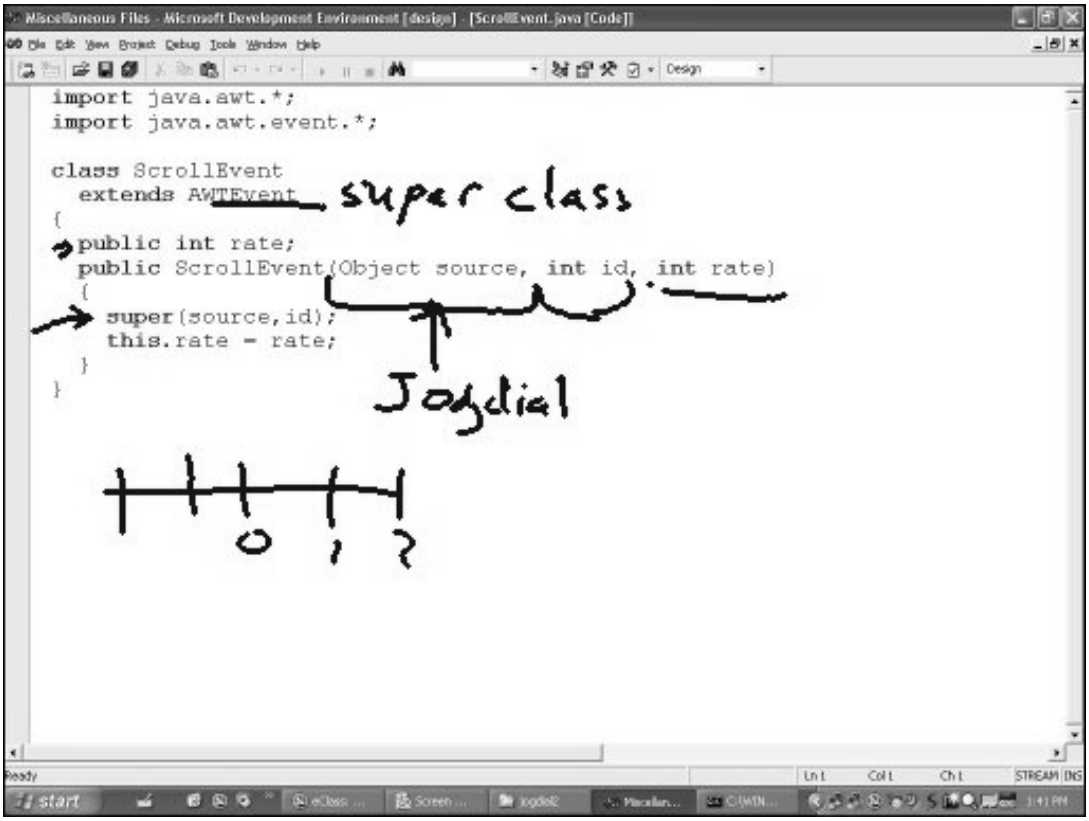


Figure 6-18: Annotated screen capture.

Finally, most instructors requested occasionally wanting to suspend audio recording. We modified the e-Board application to include a button to stop audio recording. If audio recording is not occurring, use of the button will allow instructors to resume audio recording when the e-Board observes that an audio capture service is available. The button invokes the *ControlModule* to control audio capture.

6.9.3 Student Extensions to the System

In addition to adding features to the system ourselves, we evaluated whether other, unfamiliar with INCA, could extend an existing system. We assigned a short,

three-week (20 days) project to 16 undergraduate students in the Introduction to User Interface Software course. This course introduces students to a variety of user interface software toolkits. We merely intended, as the goal with this project, to expose students to a research toolkit. The assignment was to design additional capture and access capabilities on top of the existing system, allowing for creativity and flexibility in their particular project choices. Although we did not originally intend to study this particular project, the results are important to report.

Students enter this course with varying experiences in Java. However, we use the first three short assignments to introduce students to important interface concepts while helping them learn enough Java to build applications for this course. The last three assignments, all three-week long assignments, provide students with the opportunity to work with different research toolkits that explore specific interaction concepts; these include SubArctic (http://www.cc.gatech.edu/gvu/ui/sub_arctic) for constraints, Piccolo (http://www.cs.umd.edu/hcil/piccolo/) for zoomable user interfaces, and INCA for capture and access.

Although we do not believe grades are necessarily a good measure of success, we report them here to give a ground for comparing how students performed across the final three assignments. On the first of these assignments, in which students explored building interfaces using constraints, the grades averaged 49.4 / 100 (with a standard deviation of 28.7, and 2 students did not turn in their assignments). On the second assignment, in which students explored building interfaces for capturing and accessing information, the grades averaged 74 / 100 (with a standard deviation of 24.8, and 1 student did not turn in his/her assignment). On the final assignment, in which students

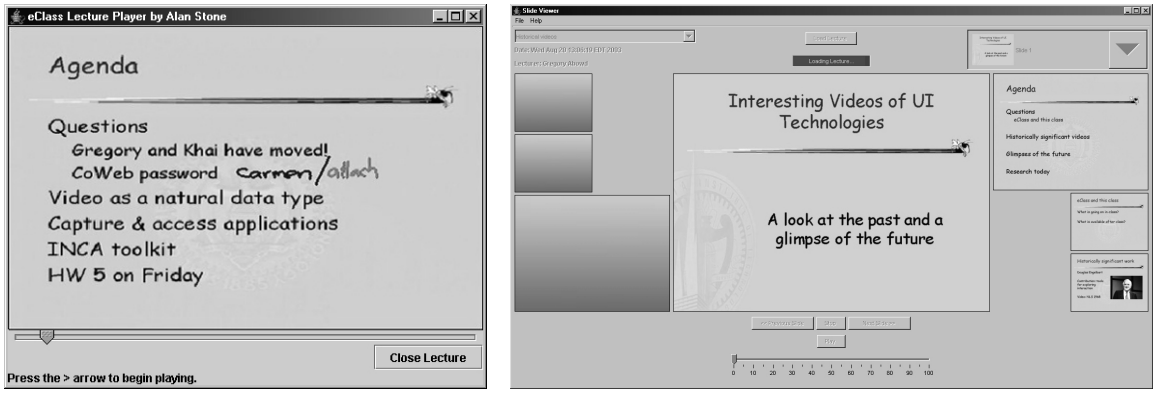
explored building zoomable user interfaces, the grades averaged 73.9 / 100 (with a standard deviation of 22.2).

For the particular assignment involving the use of the INCA toolkit, six students turned in functioning prototypes and seven other students turned in prototypes that required varying amounts of debugging. Beyond these numbers, we chose not to evaluate success and failure using hard metrics such as lines of code and development for several reasons. First, both metrics vary depending on quality and experience with coding, development process, and style. Expert programmers typically can generate code at a much faster rate than novices. Lines of code can be shortened by nesting function calls. However, depending on a programmer's style for developing applications, even expert programmers may generate large amounts of code if they choose to modularize frequently. Some developers may choose to devote more time towards architectural design than coding. Some developers may not do a code walkthrough before debugging. Without being able to monitor the development process of programmers over multiple projects involving INCA, these types of metrics would likely be inconclusive at best and misunderstood at worst.

The assignment consisted of two parts. In the first part, students generated code for a new access behavior. In the second part, students generated a design document that describes how they would build a new capture behavior. We suggested two access behaviors and one capture behavior to help students think about the kinds of features we were expecting; however, students chose to use these suggestions in their explorations. Figure 6-19 shows examples of the access interfaces developed by the students for the first half of the assignment. For the second half, most students explained how their code would be written to construct and use the *CaptureModule* and how the interface would invoke the capture function. Significantly, they described how their application would tag information so that access applications can retrieve the content later. Many students also included sketches of their interfaces.

Students, in describing how they would add capture features, displayed their understanding of how to use INCA. However, they encountered several difficulties actually producing the code that provide the access feature. These difficulties included:

- Not knowing how to work with *DataObjects*, *DataVectors* and *Serializable*



(a) (b)

Figure 6-19: Access interfaces developed by students.

objects, a problem that could be solved with minimal extra training in both INCA and Java.

- Not knowing how to generate the query for retrieving information.
- The *AccessModule* not receiving requested information from the storage component in a timely manner.

When the access interface retrieves objects, the information is returned as a *DataVector* that holds a collection of *DataObjects*. Students did not understand that a *DataVector* simply holds *DataObjects*. Additionally, students also had difficulty recovering the actual data from the *DataObject*. A *DataObject* contains a byte array and a list of *Attributes*. We provided a *Converter* object to change the byte array contained by the *DataObject* into a Java *Serializable* object. Students did not know how to work with this *Serializable* object. First they did not know how to check to see what the *Serializable* object actually represented. Secondly, they did not know how to cast the *Serializable* object back into its un-abstract form. To avoid this issue in the future, we can remove the step that requires the developer to use the *Converter* object to obtain the *Serializable* object contained by the *DataObject*.

Students also struggled to generate a query to retrieve previously captured information. These students did not build the original application and thus, often did not know which tags we used. Although we included functions to allow developers to retrieve the list of attribute types and possible values, students did not know how to use this feature. They complained about a lack of documentation. To avoid this problem in the future, we should provide students with documentation explaining how we built the

eClass prototype on top of INCA in addition to the Javadoc documentation. We also should provide a more detailed tutorial demonstrating how to use these functions. Finally, they did not understand how to construct *Query* objects themselves. We provided students with examples for constructing *Query* objects, but these were not sufficient. To avoid this problem in the future, we should make use of the operators that can be applied to a *Query* object nesting of *Query* objects easier for developers to use, because these objects play an important role in being able to retrieve the correct streams of information.

Although we provided a mechanism for developers to query for meta-content that they could then use to create a more efficient query, most students did not do what we had expected. Instead, some students constructed queries that would obtain large amounts of information from the *Repository* and then handled reduction of the data from within their applications. For example, their applications would request all the information captured by a particular instructor or request all the information captured for a particular course, even if they only want the application to show captured content from one particular lecture. Students took this approach as part of an exploratory exercise in which they just wanted to get some data for their application to use so that they can then begin to refine the application, as well as the query. This approach caused the *Repository* object to slow drastically as it tried to obtain a large amount of information to return to the client application. During large queries, the MySQL database would begin to slow down and could only serve that one connection until it completed, impeding many students' development progress. Although we frequently restarted the *Repository* to try to alleviate of this problem, many students complained

that they could not obtain information in a timely manner, if at all. Despite this problem, we were pleasantly surprised to find that six students turned in function prototypes and seven other students turned in prototypes that needed only minor debugging, with some errors only at the interface level.

Through student use (or mis-use) of the queries, we identified a major scaling concern and learned that the *StorageModule* should share time between the different access requests better than our current implementation supports. Additionally, in the current implementation, the *StorageModule* continues to process a request for information until it finishes executing that *Query*. The infrastructure forces the access application to busy wait until the *AccessModule* retrieves captured content from the *StorageModule*. To avoid these scaling concerns in the future, the infrastructure should support a way to stop the access when the query is no longer desirable. We must allow applications to tell the *AccessModule* to stop a query; in turn the *AccessModule* must be able to tell the *StorageModule* that the query should be stopped.

6.10 Summary

In this chapter, we have described how INCA has been used by us and other developers to create a number of capture and access applications, as well as to explored the capture and access design space. As we described each case study, we discussed how the developers used INCA to build the system and how the high-level features we presented in Chapter 4 helped in the construction process. INCA and applications built using the toolkit acted as a test-bed for exploring new features. We described how features of the toolkit facilitated the extension of new capture and access features on top

of some applications. A researcher also extended INCA, itself, to explore an access control mechanism to improve the security concern involved with automated capture and access application.

These case studies demonstrate that developers can use INCA to create applications that belong within the capture and access design space. For example, one of the applications built by others explored capture in the classroom, which researchers in ubiquitous computing have frequently explored. These applications captured content such as ink and video streams as many existing applications do. Furthermore, these case studies demonstrated developers can use INCA in their exploration of novel capture and access features, such as context-aware video capture or the ability control access to stored information. Finally, as we described the case studies, the various uses of the reusable components provided by the toolkit and the different application architectures across these projects demonstrate the flexibility of the toolkit (*e.g.,* see Figure 6-6, Figure 6-10 and Figure 6-14).

Table 6-1: Summary of the applications built using INCA. Applications shaded in light gray were built in part or in full by us. Applications in white were built in full by others.

Case study	Developer	How INCA facilitated the development process
Large input surface application	Computer Science Master's student with Java programming experience	Used *Repository* object, which solved storage concern.
Walden Monitor	Computer Science Master's student with no prior experience in Java programming nor GUI development	Reusable components abstracted capture/storage/access concerns & reduced task to GUI development Initial separation of concerns facilitated evolution
Classroom capture (ECE)	Computer Science undergraduate student with limited Java programming experience	Reusable components facilitated capture of information
eMeeting	Computer Science Master's student with Java programming experience & prior experience using INCA	Enabled two forms of access Capture & subscribe features used as synchronous communication mechanism
Context-aware video capture	2 Computer Science Master's students with Java programming experience	*Tagger* objects facilitated adding context-attributes to captured video (and acted as bridges to the Context Toolkit) *ObserveModule* & *ControlModule* objects facilitated the starting and stopping of video capture
Token-based access control mechanism	Computer Science Ph.D. student with Java programming experience	Architectural functions layer allowed token-based mechanism to be added through the *Registry* component
Cepher		Separation of concerns naturally supports architecture where *Registry* & *Repository* run on the Personal Server and other devices would provide capture and access interfaces
WebMemex		Context-based query mechanism enabled multiple access behaviors Attribute tagging and querying scheme was leveraged to add groupware behavior
eClass v2.0		Separation of concerns allowed for new capture & access behaviors to be added later *ObserveModule* & *ControlModule* objects facilitated addition of muting capability

CHAPTER 7

LESSONS LEARNED FROM THE INCA DEPLOYMENT

In Chapter 6, we presented case studies of different uses of INCA by us and other developers. We provided a technical description of the prototype itself and discussed how INCA influenced the design and development process of each project. In some cases, the high-level features we presented in Chapter 4 helped in the construction process. In others, the abstractions and reusable components available in the toolkit did not facilitate the development tasks or were not used as we expected.

In this chapter, we discuss the common problems we discovered from the case studies presented in Chapter 6 that will need to be addressed, as well as some important lessons that can be incorporated into the design of future architectural support for capture and access applications. These problems and insights result in the following design requirements:

- Support post-production of captured information;
- Support a more robust, exposed and extensible storage model;
- Support for communication that is not persistently connected;
- Provide platform specific versions of the toolkit, in particular one that runs efficiently on small handheld/embedded devices;
- Handle capture latency;
- Provide easier and more flexible methods for retrieving content;
- Provide easier to use components at the API level; and

- Support controlling access to captured content to ensure the security of the user's information in future applications.

7.1 Support Post-Production of Captured Information

In Section 3.2.1, we presented our reasoning behind the development of capture, storage, transduction and access building blocks for all capture and access applications, with no implied ordering of when they occur relative to each other. The case studies we presented in Chapter 6 have indicated that capture, storage and access are important concerns across these applications. Originally, we thought the need to convert captured data could be supported through transducers that operated as the application captures information.

However, developers never actually extended the *TransductionModule* inside any of the applications. Furthermore, when developers needed to convert the captured information, they typically transformed the data from one format into another at the captured session level instead of at the captured *DataObject* level as supported by the *TransductionModule* object. For example, neither the developer from Georgia Tech's School of Electrical & Computer Engineering nor the developer from the Universidade de São Paulo at São Carlos, used the original eClass system, yet in building classroom capture with INCA, they developed post-production in the same way as in eClass. After a lecture ends, the storage module restructures the captured information from that session and stores relevant information together, such as all the ink information from a particular slide. Additionally, the storage component generates images of the captured

slides and XML files that describe the captured session and point to the processed content.

We believe now that our decision to support transduction of information at the *DataObject* does not adequately meet the requirements of all applications. Although we imagine there may be reasons for continuing to support this behavior, the case studies indicate that application developers usually convert information into a different format after a captured session instead. As a result, future versions of the toolkit should include support for the post-production of captured information after a session ends.

7.2 Support a More Robust, Exposed and Extensible Storage Model

In Section 6.9.3, we described problems students encountered retrieving data from the *Repository* object for our reimplementation of the eClass system, stemming largely from misunderstanding the pre-existing system. Despite being provided with a short description of the system, students still did not fully comprehend the types of data captured by the eClass system nor how the system tagged the content. We also developed the *Repository* component as a mechanism to abstract the data management concern from the application development task. Although the abstract *StorageModule* contains functions for developers to learn about all of the *DataObjects* stored by the *Repository* and their *Attributes*, students in the class needed to create an application that programmatically invoked these features. Thus, students performed an exploratory exercise in which their applications simply retrieved data, and then they refined the designs. This practice revealed a second problem: our implementation of the *Repository* object did not scale well when multiple access clients simultaneously requested large

amounts of data.

From this case study, we learned that a storage component needs to share time between the different access requests better than our current implementation supports. Currently, the *Repository* continues to process a request for information until it finishes executing that *Query*, while the *AccessModule* busy-waits until it retrieves captured content from the *StorageModule*. Future versions of the toolkit can avoid this particular scaling concern by supporting a way for the applications to notify the *AccessModule* to stop a query; in turn, the *AccessModule* must notify the *StorageModule* that the query should be stopped. This feature would result in a more robust storage component.

To avoid developers needing to perform exploratory requests for information that could potentially result in a high performance load on the storage component, the toolkit should also include a mechanism for inspecting the data stored on the *Repository* without writing an application to do so. Developers can use this feature before programming, allowing them to understand the set of *Attributes* used and the relationship between different captured *DataObjects*.

Abstracting data management concerns from the application development task can be an extremely useful feature to novice application developers. The existence of the *Repository* component saves developers a significant amount of time because they can easily add storage functionality into their application without needing to write any code from scratch or to worry about how the application stores the information. Expert developers, however, may want more control over the way the *Repository* stores information. For example, this situation occurred in the efforts to rebuild classroom capture systems by both developers from Georgia Tech's School of Electrical &

Computer Engineering and the Universidade de São Paulo at São Carlos. Originally, they used the *Repository* object included in the INCA Toolkit to store captured information. However, each eventually customized the storage to keep relevant information together. The decision to customize the *Repository* component was made to optimize the lack of structure in the way INCA stored information. This customization in turn optimizes the access of information. From this case study, we learned that although INCA can and should provide a *Repository* object that abstracts data management concerns from the development tasks to help novice capture and access application developers, this object should also be extensible to provide expert developers with the ability to customize the database schema.

7.3 Support for Communication that is Not Persistently Connected

While developing the Cepher application, we encountered a problem with network and power consumption. Our current implementation of the network abstraction layer supports constant network connections between the client modules with the *Registry*, running on the Personal Server in the Cepher application. The constant connections consume resources heavily, most significantly network bandwidth and power, minimally available and of utmost importance on an embedded device. To resolve this problem, we modified the way components communicated with each other. We removed the network communication that tied the two components together when they both reside on the same device, bypassing the need to transmit messages through the *Registry*. We also modified the way the *Registry* communicated with remote clients. Instead of maintaining a constant network connection, a client module connects with the

Registry only when it needs to transmit a new message. This approach resulted in workable solution that allowed users to capture short notes on Tablet PCs and PDAs and to use access interfaces on those devices to retrieve the notes later. The approach would not have worked if devices needed to subscribe for captured information, meaning the application needed to acquire information immediately as capture occurs.

Constant network connections may become a problem not just with the Personal Server, but with other mobile devices as well. Furthermore, future applications may require access applications to subscribe for captured information. As a result, our implementation of the network abstraction layer needs improvement. To ensure that no messages are skipped, the *CaptureModule*, and all other modules, queue the messages in a cache until each message has been sent without error. This mechanism can be extended to support a store-and-forward messaging approach that handles the client problem of not always having a constant network connection. Additionally, instead of programming client modules to connect with the *Registry* when they need to transmit new messages to the server, the modules can be programmed to periodically connect with the *Registry*, based on a quality of service parameter. This mechanism would then also allow an access interface to subscribe for captured information. The *Registry* would queue messages to deliver to the subscriber and transmit those messages when the client module next reconnects with the *Registry*. This approach can support near synchronous communication if the client has been instructed to reconnect with the *Registry* frequently enough and asynchronous but low power and low bandwidth communication when the application can tolerate the latency.

7.4 Provide Platform Specific Versions of the Toolkit

We originally developed the INCA Toolkit to support the development of applications that would run on machines powered by workstations or laptops. Not surprisingly, this version of the toolkit does not perform well on small platforms. In addition to the network bandwidth and power issues involved with constant network connections, while developing the Cepher application, we also encountered other problems with deploying the original version of the INCA Toolkit on the Personal Server. The MySQL database we used has a large footprint and consumes much of the processing cycles on the Personal Server. INCA itself also has a large footprint.

We addressed this problem by developing a different version of INCA to run specifically on the Personal Server, known as pINCA (short for Personal INCA). This version of the toolkit used a modified communication scheme that did not require constant network connections between the *Registry* and the client modules. Instead of MYSQL, we extended the *Repository* object to use pure Java databases that have very small footprints. Finally, we repackaged pINCA such that the version placed on the Personal Server includes only the minimal number of required Java classes. If a developer needs to include specific capture and access functions on the device, she can install optional packages.

The steps taken above result in a specific version of the toolkit that runs more efficiently on the Personal Server platform than the original version. From this case study, we learned that platform specific versions of the toolkit can be developed to overcome problems that a generic solution may encounter on some class of devices.

7.5 Handle Capture Latency

While testing the context-aware video capture application, which we described in Section 6.5, the developers encountered two types of latency problems that led to incorrectly captured scene boundaries. First, the Java Media Framework requires a few seconds to initialize a camera device before it can actually record video. Secondly, the context-aware video capture application uses sensors to trigger video recording. Specifically, the location system takes time to detect people entering and leaving the room. This delay in sensing context changes and controlling the capture creates a form of latency that causes scene boundaries to not exactly align with reality.

The second form of latency can be generalized as a sensing and decision-making issue that not only computer systems designed to trigger capture may experience, but also humans as well. Furthermore, humans are very good at noticing important moments *after* they occur. For example, many parents want a record of their babies' first steps. However, because they can not always anticipate when the event will occur, they may be unprepared to take a picture when the babies actually begin to walk. Or in some situations, triggering capture, while potentially valuable, may not get highest priority if the user must tend to more urgent matters first.

From this case study, we have learned that INCA should support buffering information from capture devices, so that delays in obtaining sensor data does not result in loss of captured data. By allowing a capture device to be in a *buffering* state, rather than just simply *on* or *off,* we address both problems that result in capture latency that we described above.

7.6 Provide Easier and More Flexible Methods for Retrieving Content

In Section 6.9.3, we described the difficulties students encountered generating a query to retrieve previously captured information. Part of this problem can be resolved by making future versions of the storage component expose more about the tagging scheme used by the capture application (see Section 7.2). However, it took quite a bit of time for students to understand how to construct *Query* objects themselves. Although students had some initial trouble working with the *Query* object, when developers fully understand how to use them, they can construct very interesting and powerful queries. This is evident in the access interfaces students created after they overcame the hurdles involved in learning how to use the *Query* object as well as in other case studies, such as WebMemex.

Rather than simplifying the querying mechanism, which might decrease the ability for developers to create powerful queries, in the future, INCA should provide developers with assistance in constructing the *Query* objects themselves, by making the operators that can be applied to a simple *Query* object easier for developers to use. Secondly, we need to facilitate the construction of a compound *Query* object by nesting these objects within one another. Finally, in reading the tutorials, students understood and liked the relationship between the *Tagger* object and the *CaptureModule*. They requested the addition of a similar feature on the access side. This request motivated the later addition of the *Querier* object to the *AccessModule*, that we described in Section 4.2.3. This object allows programmers to focus on the construction of one simple *Query* object. The *AccessModule* then constructs a compound *Query* out of a collection of simple *Query* objects.

7.7 Easier to Use Components at the API Level

For nearly all the case studies we presented in Chapter 6, the developers did not develop their applications under any deadlines. As a result, these developers had time to study fully the tutorials and sample code provided with the INCA Toolkit. Stepping through these artifacts helped developers to gain enough understanding of how to use INCA to build their own applications. However, we also tested the use of INCA with a tight timetable by asking students to extend the pre-existing eClass system as part of a three-week assignment. Not given enough time to learn the API completely, students struggled to generate an efficient query to retrieve previously captured information. First, students were not provided enough detail about how the eClass application captured and tagged information. This problem made it difficult for students to know what to query. However, as we mentioned in the previous section, students also had trouble understanding how to create a compound *Query,* which involved using operators to nest and to relate multiple queries.

To avoid this problem in the future, we can provide even more detailed documentation explaining how to use INCA and additional sample code files. However, to help those working under a tight deadline and do not have enough time to familiarize themselves with such documentation, the usability of the toolkit also should also be evaluated and improved upon by viewing its API as an interface with which developers interact [Klemmer *et al*. 2004]. Making components of the toolkit easier to use requires providing an API that developers can understand and learn quickly. The case study we presented in Section 6.9.3 provides a starting point for making the components easier to use. For example, we have learned that developers have trouble working with

Serializable objects that our *DataObjects* require and provide back to access interfaces. Thus, the API can be tested and modified in the future to better meet the programmers' expectations and requirements.

7.8 Support for Controlling Access to Captured Content

In Section 6.6, we presented a case study in which a Computer Science Ph.D. student integrated a token-based access control mechanism to INCA to secure collected data for obvious privacy reasons. This addition required minor changes to the architectural functions layer, but resulted in the addition of a security feature the toolkit automatically adds to all future applications developed using INCA.

Given the other lessons described in this chapter, however, we imagine the token-based access control mechanism should be implemented differently. First, the Token Manager could distribute the tokens to the relevant capture devices, which would associate the tokens to the captured data before it sends it to the *Registry*. Second, the *StorageModule* could be modified to verify that the user has produced tokens that match those stored for the requested data. Retrieving information can take a long time, an issue we discuss further in Section 6.9.3. Thus, rather than retrieving the data only to possibly find that the user has not provided INCA with the tokens that grant her access to the data, we can eliminate this issue by having the storage component check to see that the user can access the data before pulling it from the database.

7.9 Summary

INCA facilitated the development and evolution of applications by separating the concerns involved in capture and access applications into smaller more manageable

problems providing abstractions that hide accidental features of the software that may require time and effort to overcome. Yet by the beginning of 2004, we had stopped making INCA available to other developers to use. Along with the benefits, developers also experienced unforeseen difficulties using INCA. We recognize that not all of the abstractions and separation of concerns supported by the toolkit facilitate the application development process. The infrastructure should support smaller devices, and not just the PC/server platforms. This touches upon network communications concerns, where the challenge is not to require a constant connection but rather to connect as needed for communication. This also touches upon storage concerns, where the challenge is to provide a robust and scalable solution that does not drain the resources of the device. Because we also encountered this issue in other case studies, we believe the appropriate storage model to support remains an open research question. Additionally, we need to simplify the API and provide better documentation to manage the developers understanding and expectations of the features of the toolkit. Table 7-1 summarizes these lessons. We suggest the changes that should be that can be incorporated into the design of future versions of the toolkit.

Table 7-1: Summary of the lessons learned from the different case studies. Applications shaded in light gray were built in part or in full by us. Applications in white were built in full by others.

Case study	Lessons
Large input surface application	Power of reusable components Motivated the population of library of reusable capture & access components
Walden Monitor	Separation of concerns facilitated evolution of application
Classroom capture (ECE)	Tranducers operate over *DataObjects*. Post-production support needed to operate over captured sessions Ability to add custom schemas would allow developers to tailor storage concern to specific applications
eMeeting	Capture of small chunks of data and subscription for that content resulted in a near synchronous communication mechanism
Context-aware video capture	Capture behavior needs to be modified to accommodate latency issue: • Devices take time to initiate, which can result in capture latency • Applications require time to sense and interpret context that control capture behavior. Sensor latency can result in another form of capture latency
Token-based access control mechanism	Security concern can be addressed in all future applications through support added at the architectural level
Cepher	Platform specific packages required: • Constant network connectivity leads to power consumption issue on small devices. A store-and-forward model may be more appropriate • Use of 3rd party software can lead to platform issues.
WebMemex	Sophisticated queries can result in interesting access behaviors
eClass v2.0	Queries were not intuitive construct (operators and nesting feature need to be made easier to use) *Repository* object did not scale well

CHAPTER 8

CONCLUSIONS & FUTURE WORK

This dissertation describes a conceptual framework and architecture that supports the designing, building, execution, and evolution of automated capture and access applications. In this final chapter, we summarize our research and briefly describe some areas that merit future research.

8.1 Research Summary

In Chapter 1, we presented definitions of automated capture and access. We described the value automated capture and access provides to humans in their daily lives. Chapter 2 provided a summary of the related work in the area of automated capture and access, presenting applications that demonstrate the potential benefits of services that record information on behalf of the user and making that content available for later review.

In Chapter 3, we described our experiences with developing and extending the original eClass application with additional functionalities, which required a significant amount of time and effort that could have been focused on exploring and solving human-computer interaction level issues. The specific systems level challenges we encountered were caused by the fact that:

1. Capture and access applications are built as an *ad hoc* confederation of components. Developers can not determine *all* of the features of a system at design time. As seen with the Zen* system, capture and access

applications must be extended to include devices and components as additional functionalities become necessary.

2. Not only can developers not pre-determine all of the functionality for a capture and access application, but perhaps more importantly, they can not pre-determine all of the types of captured information, because applications are comprised of an *ad hoc* confederation heterogeneous devices and components. As a result, a rigid hierarchical storage scheme must be modified and made more flexible.
3. Common architectural concerns exist across applications but were often rebuilt for each system. These concerns can be abstracted as reusable components to minimize time and effort spent reengineering these solutions.
4. Application code is often tightly coupled with the underlying network communication scheme. We can not easily modify the application code nor the communication scheme without risking adverse changes to the other. However, the design of a software application must be iterated upon during its design, development and deployment phases.

We used the lessons from addressing these challenges to form the basis for a generalized high-level architecture for capture and access applications. We derived capture, storage, transduction and access building blocks for all capture and access applications, with no implied ordering of when they occur relative to each other. Furthermore, the integration of information should be performed when information is

being accessed. This framework decouples the underlying communication structure from the essential application features, effectively abstracting networking concerns from the development process. We then used key features of the architectural framework to identify a *focused* design process that abstracts common, accidental development tasks and allows developers to address the interaction design and the information design issues specific to a software application.

In Chapter 4, we discussed our implementation of the INCA Toolkit. We presented four high level features supported by the toolkit that facilitate the development of this class of application. In Chapter 5, we provided a short example that demonstrates how the INCA toolkit can be used to develop simple capture and access applications, in this case one that continuously buffers audio in the local environment to a user for 15 minutes. This application allows the user to review audio information from the recent past.

In Chapter 6, we presented case studies that demonstrate how the INCA Toolkit was used by us and others to build a variety of applications. We also described a structured case study, in which we reimplemented the eClass/Classroom 2000 system and studied the ability of others to build applications on top of a system previously developed using INCA. In each case study, we presented what the developers built and how INCA was used in the development task. In Chapter 7, we summarized the success and challenges encountered by the developers and discussed how to iterate upon the design of INCA based on the observed success and failures of these projects.

We concluded by identifying and supporting a software structure that separates the core functionalities of a capture and access application and provides additional

germane features and abstractions. This infrastructure encourages a focused model for designing, implementing and evolving automated capture and access applications. It should include a collection of reusable components that address the core capture, access, storage, transduction and post-production concerns of this class application. Additionally, the architecture should support the systematic association of meta-data to content, context-based queries and garbage collection, and provide the ability to observe and control the run-time of a system. Finally, the infrastructure should hide from novice developers the complexities involved with the communication structure and data management aspects of the system, while allowing the expert developers to easily customize the data management scheme if necessary. The contributions of this thesis are:

- The identification of the capture and access design space;
- The identification of a conceptual framework and a focused design process that addresses the minimal set of requirements for constructing and evolving capture and access applications;
- The implementation of an infrastructure supporting the above framework and design process to act as a testbed for investigating problems in automated capture and access;
- The population of a toolkit with common, reusable capture and access services; and
- The exploration of the capture and access design space, by us and other developers, using the toolkit.

8.2 Future Research Directions

We believe the application development issues discussed in Chapters 6 & 7 indicate some requirements that should be addressed in subsequent versions of the INCA toolkit:

- Support post-production of captured information;
- Support a more robust, exposed and extensible storage model;
- Support for communication that is not persistently connected;
- Provide platform specific versions of the toolkit, in particular one that runs efficiently on small handheld/embedded devices;
- Handle capture latency;
- Provide easier and more flexible methods for retrieving content;
- Provide easier to use components at the API level; and
- Support controlling access to captured content to ensure the security of the user's information in future applications.

Beyond these requirements, the area of automated capture and access has other relevant problems that have not been explored fully. We discuss some issues that warrant investigations in this domain in the following section.

8.2.1 Quality of Service

To sustain tolerable performance, by default, the INCA Toolkit captures information in low-bandwidth and uncompressed data formats instead of high quality content. However, developers can also set the capture fidelity/quality specific to the

needs of their applications. However, fidelity is only one type of quality of service metric. The INCA toolkit needs to be extended to support other metrics, including access latency and coverage.

Access latency relates to the amount of time after information has been captured that is required before the access interface can obtain the content. As we mentioned, in the current implementation, capture at low bandwidth quality ensures that access clients can obtain information in a timely manner. However, by capturing at a low fidelity, information is also stored at a low quality. Instead, the toolkit should allow access components to specify the latency at which the information needs to be delivered and compression can be applied accordingly.

By coverage, we mean how much information needs to be recorded. In some situations, full video recording may not be possible but other media or streams can still be captured. These situations can be dictated by user preferences or by environmental constraints. When these situations occur, rather than not having any recording take place, the capture components should alert subscribers to other options currently supported. Automatic negotiation can occur between the components or the user can be brought into the loop to resolve conflicts.

8.2.2 Sophisticated Access Interfaces

The value of automated capture applications is apparent only when users later need to review the content. Being able to recover the captured content is an important requirement. Unfortunately, most applications still support very simple access behaviors, such as indexing and playback. Assuming capture applications continuously

record information over long periods of time, users will need help to browse through mass quantities of stored content. Automatic summarization of dynamically retrieved captured content, for example, would be a useful access technique, as well as an interesting research problem to investigate. Additionally, we have previously built an interface, known as the Space-Time Browser (see Figure 8-1), to visually represent and enable search for content using two naturally salient clues: time and location [Chiu and Truong 2002]. Generic search mechanisms, such as this, can be added to the toolkit, to allow application developers to add more quickly searching capabilities to their applications without needing to build the interfaces to support that behavior.

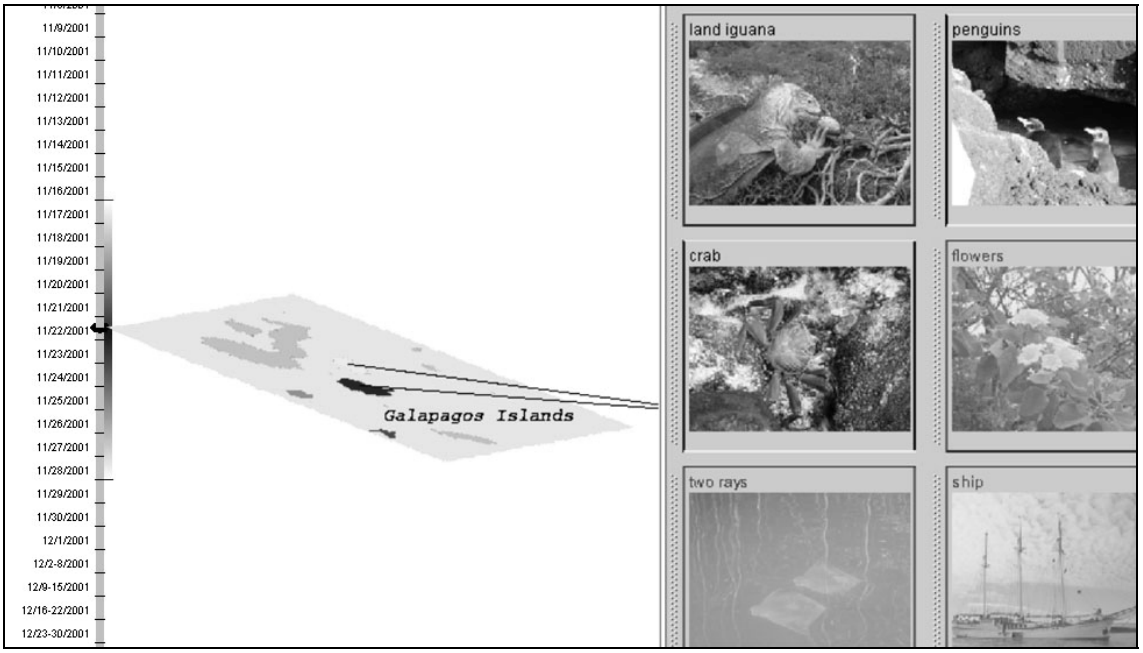


Figure 8-1: The Space-Time Browser interface.

8.2.3 End-User Specification of Capture & Access Behaviors

Ubiquitous computing technology for domestic environments is becoming an increasingly prominent theme of research to support the needs of families and individuals in their homes. As the trend towards technology-enriched home environments progresses, the need to enable users to create applications to suit their own lives increases. We want to enable non-programmers to create services easily that meet their needs, but we do not want to require much programming experience on the part of these end-users. Formative user studies indicate that user descriptions of a service tend to focus on the function and not the devices. Although INCA offers an architectural model that may simplify the development process, that model does not match how users naturally think about applications.

We have performed a preliminary design and evaluation of a GUI interface, known as CAMP (short for Capture and Access Magnetic Poetry) that allows users to map their conceptual models onto the INCA architectural model [Truong *et al.* 2004].

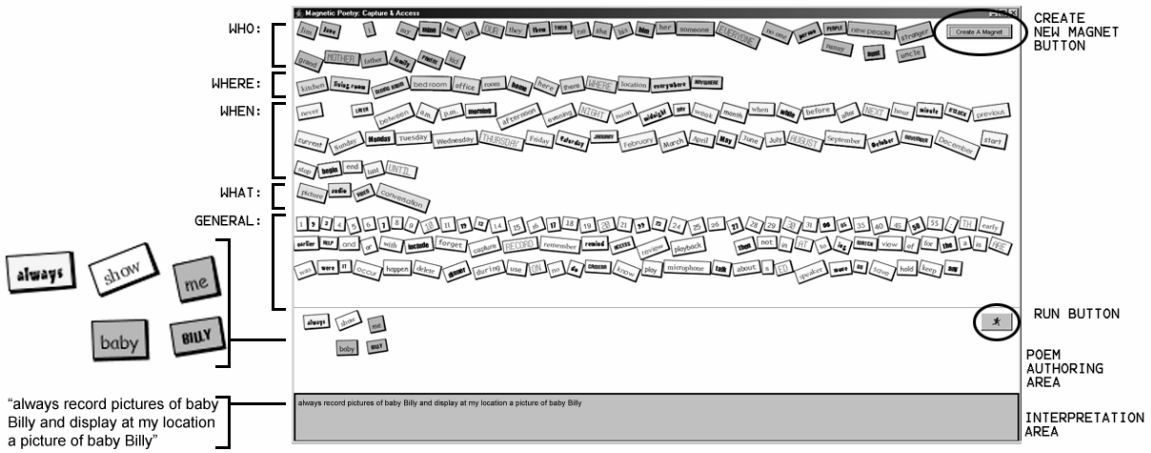


Figure 8-2: The Capture & Access Magnetic Poetry interface.

The CAMP interface offers users a flexible way to specify desired applications through the use of a "magnetic poetry" metaphor. Users can combine home or capture-themed magnetic poetry pieces into statements that describe an application (see Figure 8-2). Our early evaluation indicates that the magnetic poetry interface is simple to learn and to use, and allows users to specify the types of applications they want in the way that makes the sense to them. However, we believe other effective methods allowing end-user specification of ubiquitous computing applications remain, and need to be developed and studied.

8.2.4 Support for Socially-Appropriate Capture

Manual recordings, even those augmented by computing, take the user out of the moment and are less likely to garner valuable information. Always on capture reduces many of these problems. However, this approach often has been met with resistance from users and the research community alike. Concerns about automated capture in sensitive domains, such as the home, often center on privacy and security of information, the unnecessary archiving of unmanageable amounts of data, and the placement and locations of capture devices and storage of recorded information. Although users tend to object to the instrumentation of capture devices when simple sensors are sufficient [Beckmann *et al.* 2004], Melenhorst *et al.*'s study shows that people are willing to accept intrusive technologies that offer useful services [Melenhorst *et al.* 2004]. Thus, research to balance the social, technical, and practical concerns of capture applications warrants further investigation.

To support capture in a sensitive domain, such as caring for CWA, we have

developed experience buffers, a collection of continually active capture services that together comprise a capture architecture embedded in an environment [Hayes *et al.* 2005]. However, experience buffers do not inherently archive information. The experience buffer architecture includes a mechanism to discover available services in an environment that uses a local short range *ad hoc* wireless network (*e.g.* Bluetooth). There is also a protocol for specifying to these services the portions of the buffer to archive. As a result, experience buffers provide many of the benefits of keeping capture services under human control while providing the benefits of always on capture. By using buffers, users can be assured of getting all of the content they want and very little of what they do not, significantly reducing the access and analysis time when the information is needed later. Buffers can be provided in environments where human concerns, including cost, make a large array of always on capture services impractical. Finally, buffering inherently addresses the latency issues described in Section 6.5.2.

We will deploy this approach in public and semi-public settings and study its adoption. If our studies confirm that this technique provides a socially appropriate mechanism for capture, we will integrate this approach into future versions of the toolkit. However, until the research community has studied, designed and addressed the problem of capturing in a socially-appropriate manner, automated capture may continue to be viewed as invasive, recording and archiving data at all times without human control, producing much more content than necessary.

8.2.5 Automatic Negotiation of Privacy Policies

The construction of particular situations and the concepts of capturing details

about them influence policies users may have about capture. In some situations, users may not have the opportunity to negotiate whether or not to disable capture features. Other times, the users may have already undergone this step previously and prefer to default to their past chosen levels of consent. We believe providing a mechanism to support this negotiation is an interesting social and technical problem. All user preferences can be generalized into unique privacy policies. Users can carry beacons that transmit their preferences to the environment or to other devices, which can automatically negotiate the appropriate behavior given the preferences of surrounding users, only involving the users when no appropriate resolution can be reached. Such a system might also learn from human intervention and apply this knowledge in future scenarios.

8.2.6 Capture-Resistant Environments

With the ubiquity of small mobile capture devices, such as camera phones, it is now possible to capture information anywhere, raising a legitimate concern for many organizations and individuals. Although legal and social boundaries can curb the capture of sensitive information, it sometimes is neither practical nor desirable to follow the option of confiscating the capture device from an individual. Techniques for preventing capture without requiring any cooperation on the part of the capture device or its operator must be developed.

We have developed proof of concept implementation of a capture-resistant environment that prevents CCD and CMOS cameras from recording of still and moving images [Truong *et al.* 2005]. Our solution involves a simple tracking system that uses

computer vision for locating any number of retro-reflective CCD or CMOS camera sensors in a protected area. Specifically, we use a Sony Digital HandyCam video camera placed in *NightShot* mode. IR transmitters surround the lens and a narrow bandpass IR filter covers the detector's lens. This instrumentation, referred to as the *detector*, projects an IR light beam outwards from the camera and detects any retro-reflective surfaces within the field of view. The specific placement of the IR illuminator around the perimeter of the detector's lens ensures a bright retro-reflection from cameras within the field of view of the detector. A pulsing light is then directed at the lens, distorting any imagery the camera records. Although the directed light interferes with the camera's operation, it can be designed to impact minimally the sight of other humans in the environment. Specifically, we pair a projector of 1500 lumens with our camera detector. The projector emits localized light beams of an area slightly larger than the size of the reflection. This approach also requires no cooperation on the part of the camera nor its owner.

The above approach, however, only works against consumer level CCD and CMOS cameras. Much more research needs to be conducted in this area to achieve success in preventing all cameras from recording in protected area. Similarly, techniques for preventing capture of audio without requiring any cooperation on the part of the recording device or its operator must be developed.

8.3 Conclusions

This thesis dissertation presents the requirements for and an implementation of an architecture for supporting the design, implementation, and evolution of capture and

access applications. The architecture supports a focused design process for building applications that encourages designers to decompose the problem into core concerns: capture, storage, access and transduction. We provide an infrastructure/toolkit that helps to translate these designs into executable form. The toolkit provides additional support for addressing other concerns such as removal of content, and the ability to observe and control capture and access behaviors. The work also allows the exploration of interesting research issues within the area of automated capture and access.

APPENDIX A

INCA IMPLEMENTATION DETAILS

A.1 The Network Abstraction Layer

Although the current version of INCA uses a client-server network configuration, we also experimented with using a peer-to-peer network architecture implementation in earlier versions of the INCA Toolkit. Although this meant minor differences in the way we implemented the architectural building blocks, we still achieved the same high-level results. Our implementation of the peer-to-peer configuration failed in actual deployment settings because it required all clients to have unique IP addresses. Remote clients experienced problems communicating to devices connected to the network through a router, because those devices had only local IP addresses assigned by the router that clients outside the local network were not able to resolve. The router can be configured to forward ports to the appropriate devices (and thus perhaps the appropriate modules), however, this approach does not scale. Furthermore, this approach assumed administrative access to the router which may not always be available to developers.

For practical reasons, we developed all the architectural functions described in Section 4.2 above a custom client-server network architecture implementation, referred to as the network abstraction layer. In this section, we describe our implementations of the reusable components included in this abstraction layer. We also present the programming interface supported by each component. This discussion facilitates the

subsequent discussion in this Appendix about how we implemented the *Registry* and the reusable functional building blocks.

A.1.1 The Server

In our client-server network architecture implementation, the *Server* object binds to two ports. The *Server* uses one of the ports to support synchronous communication between it and client socket connections. The second port supports asynchronous communication.

Application developers can extend this object in their code. As developers of the INCA Toolkit, we extend this object to create a *Registry* server that maintains the list of available functional modules and coordinates communication between them. The *Server* object has the following programming interface:

```
Server()
Server(int port_1)
Server(int port_1, int port_2)
```

Developers can instantiate a *Server* object using any of the three constructor methods above. The developer can specify the two available port numbers that the *Server* object can use. If only `port_1` is provided, the *Server* object automatically attempts to bind to the next port number. If no port values are specified, then the *Server* object attempts to bind to the default port values (6789 and 6790).

`serverStateChanged(boolean state)`

When the *Server* object successfully binds to two ports, it invokes this callback method and passes in a true value as the `state` parameter. When the *Server* loses its binding to the ports or when it experiences any failure, it invokes this callback method and passes in a false value as the `state` parameter.

`clientConnectionFormed(Connection connection)`

The *Server* object continuously performs a busy wait until it accepts a new socket connection to the first port. It then automatically accepts a new socket connection to the second port. Next, it constructs *Connection* objects using these *Socket* objects. Finally, it invokes this callback method, passing into it the *Connection* object as the `connection` parameter.

`clientConnectionBroken(Connection connection)`

When the *Server* loses its connection to a client, which typically happens when the client closes its socket connections with the *Server*, it invokes this callback method, passing into it the corresponding

Connection object as the `connection` parameter.

A.1.2 Connections

A *Connection* object supports communication between a client and the server using two socket connections. The *Connection* object uses the first socket connection for synchronous communication between the client and the server. It uses the second socket connection to support asynchronous communication.

Application developers can use the *Connection* object to act as a network client. From the server side perspective, application developers can use the *Connection* object to send data to the client. As developers of the INCA Toolkit, we extend this object to create the reusable functional building blocks described in Section 4.2. Additionally, the *Registry* object coordinates communication between the different client modules by systematically writing and reading data from the different *Connection* objects based on the message types initiated by the clients. The *Connection* object has the following programming interface:

```
Connection(String host)
Connection(String host, int port_1)
Connection(String host, int port_1, int port_2)
Connection(Socket socket_1, Socket socket_2)
```

Developers can instantiate a *Connection* object using any of the four constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Server* object with

which the *Connection* object should connect. If only `port_1` is provided, the *Connection* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, the *Connection* object attempts to connect with the default port values (6789 and 6790).

Developers can also instantiate a *Connection* object with two previously constructed *Socket* objects. We use this final method for instantiating a *Connection* object, in our implementation of the *Server* object. As previously described, the *Server* object makes *Connection* objects available to developers who wish to program custom communication behaviors between the server and the client. The *Server* continuously performs a busy wait until it accepts a new *Socket* connection to its first binded port. It then automatically accepts a new *Socket* connection to the second binded port. Finally, the *Server* constructs a *Connection* object using these *Socket* objects.

`write(byte[] data)`

The `write` method allows a client or server application to transmit data synchronously to its counterpart. The method busy-waits until data transmission have been attempted.

`byte[] read()`

The `read` method allows a client or server application to perform an explicit read for data transmitted synchronously by its counterpart. The method busy-waits until data become available. This method returns the data as an array of bytes.

`sendData(byte[] data)`

The `write` method adds the array of bytes a client or server application wants to send to its counterpart onto a data queue and then later transmits that data asynchronously.

`dataReceived(byte[] data)`

When the *Connection* object receives data transmitted over the asynchronous socket channel, it invokes this callback method and passes the data into the method as a parameter. This method means the *Connection* object notifies the client or server application when has been read, rather than forcing the client or server application to busy-wait on `read` operation.

`disconnected()`

When the *Connection* loses its connection to a client or a server, which typically happens when the counterpart closes the socket connections, it invokes this callback method.

`kill()`

Developers can close the network connections between the client and the server using this `kill` method.

A.2 The Architectural Functions Layer

INCA includes the reusable *Connection* and *Server* objects provided in the network abstraction layer in the *Registry* server and all the functional client modules described in Section 4.2. We use the synchronous communication channel available in our implementation of the reusable network components to support handshakes and other types of messages that require a client *Connection* to busy-wait for the response, such as when an *AccessModule* requests information. The asynchronous channel supports all other traffic.

Although we originally designed INCA components to exchange network messages structured in XML format, we ultimately opted to use a custom message structure because:

1. Despite existing support for creating XML messages that hold binary objects, these tools did not always successfully recover the original binary objects from the XML message; and
2. XML messages still proved to be fairly slow for applications to parse

The client modules and the *Registry* server transmit messages between one another in ASCII format. The first line of each message contains the String message type header. Depending on the message type, the message may contain additional information, such as *Serializable* objects converted into a Base-64 encoded ASCII string.

In this section, we describe our implementation of the reusable components included in this architectural functions layer as well as the *Registry* server. We also present the programming interface supported by each component.

A.2.1 The Registry Server

The *Registry* object maintains a list of the available functional modules and coordinates communication between all the different modules. The *Registry* object extends the *Server* object provided by the network layer described in Appendix A.1.1. When a client module connects with the *Registry*, the server generates a unique identifier (UID)[3] for that network connection. The *Registry* generates a UID value by

[3] In Section 4.1.3, we discussed the role of UID values in observing and controlling the run-time state of the system.

concatenating the client module's IP address with the connection type and the timestamp of when the module connected with the *Registry*. Next, the module informs the Registry about its architectural function (capture, access, or storage, *etc.*). For each client module, the *Registry* server then creates a *ConnectionHandler* object that reads and handles all messages sent by the client module.

We developed the *Registry* object as a self-contained component, not meant to be extended or modified by application developers. To allow developers to create and to use a *Registry* within their application, however, the object contains the following programming interface:

```
Registry()
Registry(int port_1)
Registry(int port_1, int port_2)
```

Developers can instantiate a *Registry* object using any of the three constructor methods. The developer can specify the two available port numbers that the *Registry* object can use. If only `port_1` is provided, the *Registry* object automatically attempts to bind to the next port number. If no port values are specified, then the *Registry* object attempts to bind to the default port values (6789 and 6790).

A.2.2 The CaptureModule

The *CaptureModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to

capture and make content available to other parts of the system.

The *CaptureModule* and all other client modules include watchdog threads that monitor their network connections with the *Registry*. When the connection breaks, the thread continuously attempts to reconnect every 5 seconds (an arbitrary number) with the *Registry* until it succeeds. To ensure that no messages are skipped, the *CaptureModule*, and all other modules, queue the messages in a cache until each message has been sent without error.

The *CaptureModule* object has the following programming interface:

```
CaptureModule(String host)
CaptureModule(String host, int port_1)
CaptureModule (String host, int port_1, int port_2)
```

Developers can instantiate a *CaptureModule* object using any of the three constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *CaptureModule* object should connect. If only `port_1` is provided, the *CaptureModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, then the *CaptureModule* object attempts to connect with the default port values (6789 and 6790).

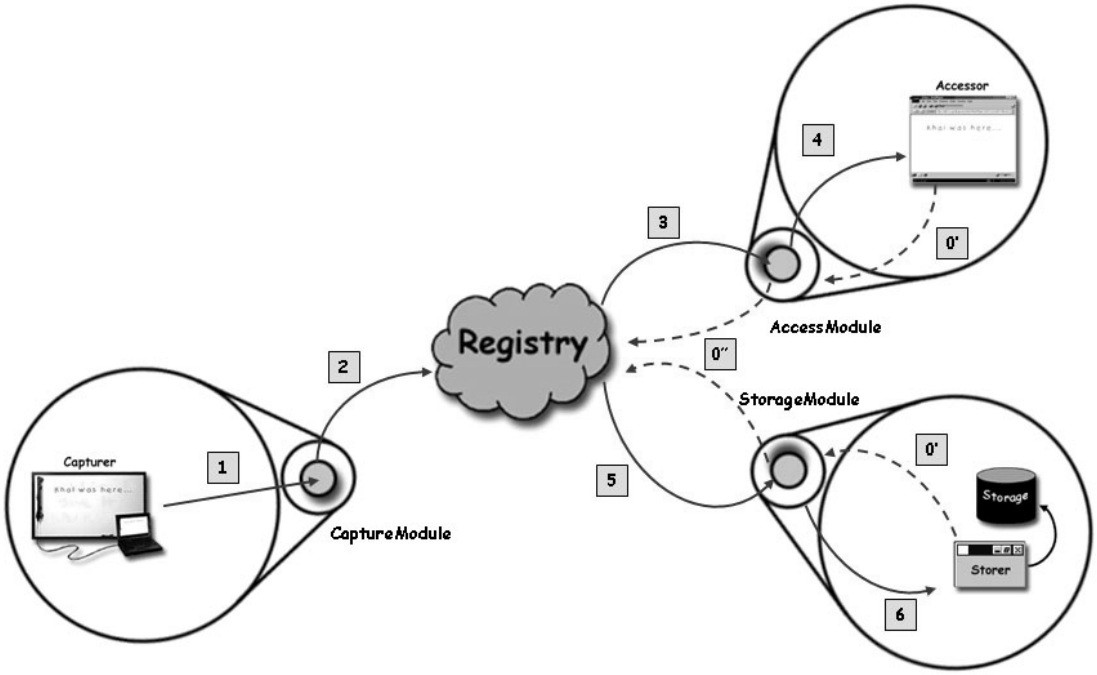


Figure A-1: The capture of information. (1) The capture application invokes the `capture` method with a *DataObject* holding the captured content as a parameter. (2) The *CaptureModule* sends to the *Registry* a `CAPTURED_DATA` message that includes the *DataObject* encoded as a Base-64 ASCII string. (3 & 5) The *Registry* relays the message to client modules that are interested in the captured content. (4 & 6) The *AccessModule* and the *StorageModule* make the *DataObject* available to the access interface and the storage application respectively. (0' & 0") In order to receive captured information, the access and storage clients must subscribe for the content prior to the capture activity.

`DataObject capture(DataObject data)`

The `capture` method allows a client application to make data available to other parts of the system (such as access, storage, *etc.*). When an application invokes this method, the *CaptureModule* traverses the list of registered *Tagger* objects to gather a list of meta-data to associate with the captured *DataObject*. Once this process completes, the *CaptureModule* sends to the *Registry* a `CAPTURED_DATA` message containing the *DataObject* encoded as a Base-64 ASCII string. After the *Registry* has

read the message, it relays the message to any *AccessModule*, *StorageModule* or *TransductionModule* interested in the content. The method returns the *DataObject* made available to other parts of the system. This object includes the meta-data generated by the *Tagger* object.

```
DataObject update(DataObject old_data,
                  DataObject new_data)
```

The `update` method allows a client application to make updated data available to other parts of the system (such as access, storage, *etc.*). This method functions similarly the `capture` method. The primary difference lies in the `UPDATED_DATA` message it transmits to the *Registry*. The message contains the previously captured *DataObject* encoded as a Base-64 ASCII string, followed by the updated version of that information.

```
addTagger(Tagger tagger)
```

The `addTagger` method allows a client application to register a *Tagger* object that will associate a specific kind of meta-data to the captured information.

`removeTagger(Tagger tagger)`

The `removeTagger` method allows a client application to unregister a *Tagger* object.

`stateChanged(boolean state)`

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.3 The StorageModule

The *StorageModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to obtain captured information that needs to be stored. Application developers must extend this object with back-end storage and retrieval support. We include in the INCA Toolkit two extensions of the *StorageModule* for application developers to instantiate and use within their application. The *Repository* and *FileRepository* objects provide storage and retrieval support using a MYSQL database and a simple hierarchical file structure respectively.

The database schema used by the *Repository* object consists of three tables. The first table holds the *DataObjects* (as blobs) and an ID value for each entry. The second

table holds the list of *Attributes*, with columns for the attribute name, the attribute value, and an ID value for each entry. The third table holds the associations between the attribute IDs and the data IDs; and two additional columns exists to hold the start and stop time of when the attribute is true for the captured data. If, for example, an application captures video in four minute segments and annotates the video with people present within each scene, a person may be present and then moves out of the field of view during the scene. These two columns allow the application to specify that a person was present in specific portions of the video.

The decision to use an SQL database as a back-end storage component does lead to some deployment issues; mainly, requiring the availability or installation of the SQL server. As a result, we have also implemented a *FileRepository* object that provides the exact functionality using a hierarchical file structure. The *FileRepository* uses three files that hold essentially the same information as the three tables used in our SQL implementation. Instead of storing the *DataObjects* within the file that acts as the data table, this file instead stores the locations of the *DataObjects* in the file system. The *FileRepository* actually stores the *DataObjects* on disk and uses folders to manage the exploding number of files in a single folder, a problem that can affect the time to list and fetch information from disk. The folder structure consists of the year, month, date, hour and minute at which the data was captured and time-stamped.

Application developers can use the *Repository* or *FileRepository* objects when they want the data storage concern abstracted from their software development task. However, if application developers wish to use other back-end storage software or have specific ways they want the information to be stored, they can extend the

StorageModule object by defining the follow seven methods `store`, `update`, `retrieveData`, `retrieveRelatedAttributes`, and `retrieveAttributesClusters`, `getAttributeNames`, and `getAttributeValues`. The *StorageModule* invokes these methods when it receives requests from other parts of the system. Overall, the *StorageModule* object has the following programming interface:

```
StorageModule(String host)
StorageModule(String host, int port_1)
StorageModule(String host, int port_1, int port_2)
```

Developers can instantiate a *StorageModule* object using any of the three constructor methods above. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *StorageModule* object should connect. If only `port_1` is provided, the *StorageModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, then the *StorageModule* object attempts to connect with the default port values (6789 and 6790).

```
store(DataObject data)
```

When the *StorageModule* object receives a `CAPTURED_DATA` or `TRANSDUCED_DATA` message, which contains information encoded

as a Base-64 ASCII string that needs to be stored, it invokes this callback method with that information converted back into a *DataObject* as the parameter. When application developers extend the *StorageModule* object, they must define this method to store information.

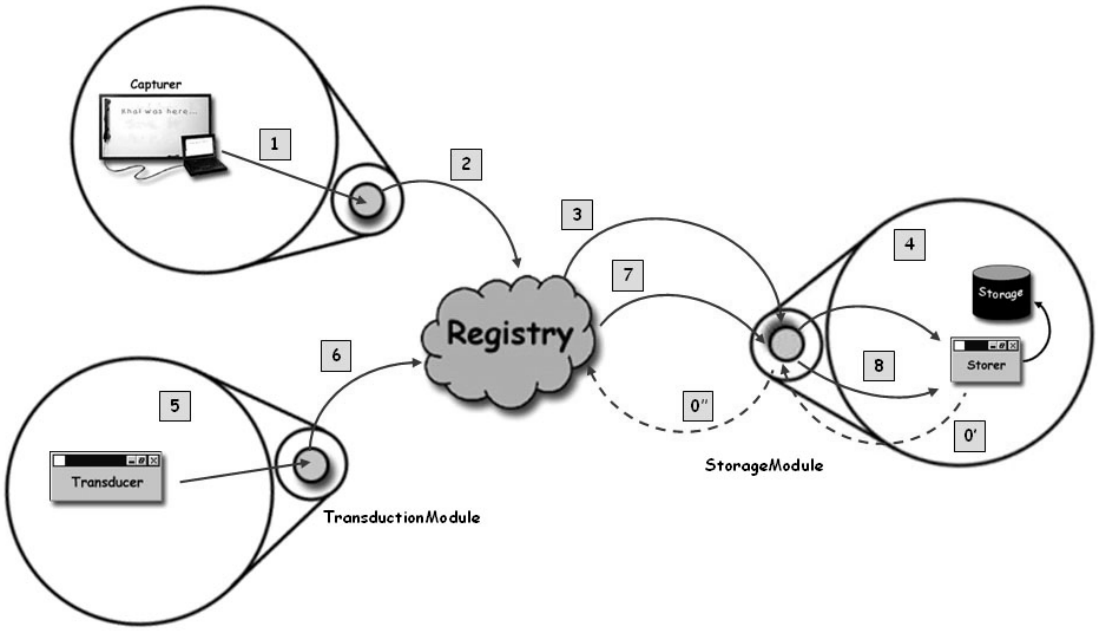


Figure A-2: The storage of information. (1) The capture application invokes the *CaptureModule*'s `capture` method with a *DataObject* holding the captured content as a parameter. (2) The *CaptureModule* sends to the *Registry* a `CAPTURED_DATA` message that includes the *DataObject* encoded as a Base-64 ASCII string. (3) The *Registry* relays the message to the *StorageModule* that is interested in the captured content. (4) The *StorageModule* invokes its `store` callback method with the captured information converted back into a *DataObject*. (5, 6, 7 & 8) In a similar manner, the *Registry* receives and relays transduced information to the *StorageModule*, which then stores the data. (0' & 0") In order to receive the `CAPTURED_DATA` message, the access storage must subscribe for the content prior to the capture activity. (0' & 0") In order to receive captured and transduced information, the storage client must subscribe for the content prior to the capture and activities.

```
update(DataObject old_data, DataObject new_data)
```

When the *StorageModule* object receives an `UPDATED_DATA` message, which contains the previously captured *DataObject* encoded as a Base-64 ASCII string, followed by the updated version of that information, it invokes this callback method with that information

converted back into *DataObject* format as the parameters. When application developers extend the *StorageModule* object, they must define this method to update information.

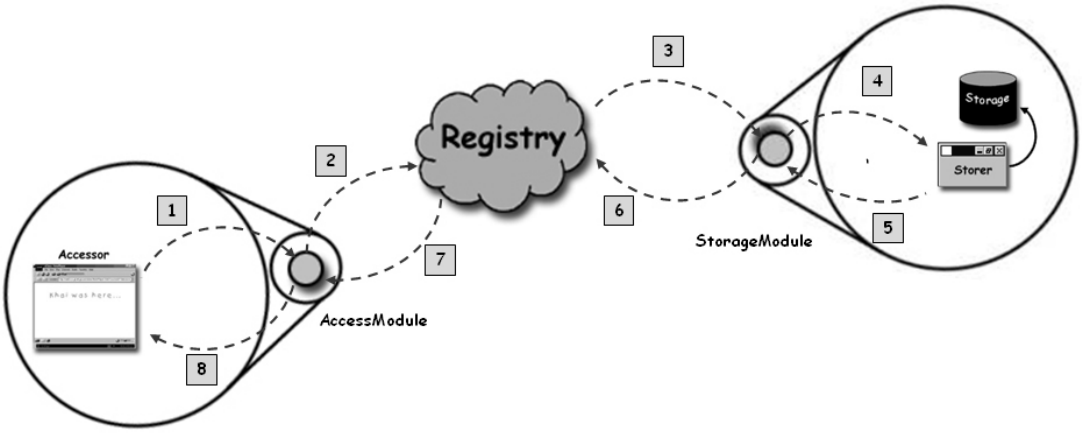


Figure A-3: The retrieval of information. (1) The access interface invokes the *AccessModule*'s `request` method with a *Query* specifying the content to retrieve as a parameter. (2) The *AccessModule* sends to the *Registry* a `DATA_REQUESTED` message that includes the *Query* encoded as a Base-64 ASCII string. The *AccessModule* busy-waits for a returning `STORED_DATA` message. (3) The *Registry* relays the message to the *StorageModule* that may have the captured content. (4 & 5) The *StorageModule* invokes its `retreiveData` callback method with the request converted back into a *Query*, which returns a *DataVector*. (6) The *StorageModule* sends back to the *Registry* a `STORED_DATA` message that includes the *DataVector* encoded as a Base-64 ASCII string. (7) The *Registry* relays the message back to the *AccessModule* requesting the content. (8) The *AccessModule* returns the requested information as a *DataVector*.

`DataVector retrieveData(Query query)`

When the *StorageModule* object receives a `DATA_REQUESTED` message, which contains a *Query* encoded as a Base-64 ASCII string, it invokes this callback method with that information converted back into a *Query* as the parameter. When application developers extend the *StorageModule* object, they must define this method to retrieve data from the back-end storage component and returns the information as a *DataVector*, a *Vector* of *DataObjects*. The *StorageModule* sends to the

Registry a `STORED_DATA` message containing the *DataVector* encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays the message to the *AccessModule* requesting the information.

`AttributeVector retrieveRelatedAttributes(Query query)`

When the *StorageModule* object receives a `RELATED_ATTRIBUTES_QUERY` message, which contains a *Query* encoded as a Base-64 ASCII string, it invokes this callback method with that information converted back into a *Query* as the parameter. When application developers extend the *StorageModule* object, they must define this method to determine the list of attributes for the stored information that matches the specified *Query*. For example, when the *StorageModule* receives a request for the list of related attributes for content captured in Atlanta, if the storage component has a picture of John and Jane Doe taken in Atlanta, then the list of related attributes should indicate that the content is about John Doe and Jane Doe. The *StorageModule* sends to the *Registry* a `RELATED_ATTRIBUTES_RESULT` message containing the *AttrbitueVector* encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays the message to the *AccessModule* requesting the information.

`Vector retrieveAttributesClusters(String attribute_type)`

When the *StorageModule* object receives an `ATTRIBUTES_CLUSTERS_QUERY` message, which contains a *String* of the encoded as a Base-64 format, it invokes this callback method with that information converted back into a Java *String* as the parameter. When application developers extend the *StorageModule* object, they must define this method to find the related attributes for all possible values of the specified attribute type. For example, when the *StorageModule* receives a request for the clusters of related attributes for content captured tagged with location information, if the storage component has a picture of John Doe taken in Atlanta during May and a picture of Jane Doe taken in San Francisco during September, then the clusters of related attributes should indicate that the storage component has content captured in Atlanta about John Doe in May and content captured in San Francisco about Jane Doe in December. The *StorageModule* sends to the *Registry* an `ATTRIBUTES_CLUSTERS_RESULT` message containing a *Vector* of *AttributeVectors* encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays the message to the *AccessModule* requesting the information.

`Vector getAttributeTypes()`

When the *StorageModule* object receives an `ATTRIBUTE_TYPES_REQUESTED` message, it invokes this callback method. When application developers extend the *StorageModule* object, they must define this method to determine the list of attribute types used to describe the stored content. For example, when the *StorageModule* receives a request for the list of attribute types, if the storage component has a picture of John Doe taken in Atlanta during May, the list of attribute types should contain location, time and people present. The *StorageModule* sends to the *Registry* an `ATTRIBUTE_TYPES_RESULT` message containing a *Vector* of attribute type strings encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays the message to the *AccessModule* requesting the information.

`Vector getAttributeValues(String type)`

When the *StorageModule* object receives an `ATTRIBUTE_VALUES_REQUESTED` message, it invokes this callback method. When application developers extend the *StorageModule* object, they must define this method to list all the attribute values associated to a given attribute type. For example, when the *StorageModule* receives a request for the list of attribute types, if the storage component has a picture

taken in Atlanta and a picture taken in San Francisco, the list of attribute values for the location attribute type should include Atlanta and San Francisco. The *StorageModule* sends to the *Registry* an `ATTRIBUTE_VALUES_RESULT` message containing a *Vector* of attribute value strings encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays the message to the *AccessModule* requesting the information.

`gc(Query query)`

When the *StorageModule* object receives a `GARBAGE_COLLECT` message, which contains a *Query* encoded as a Base-64 ASCII string, it invokes this callback method with that information converted back into a *Query* as the parameter. When application developers extend the *StorageModule* object, they must define this method to discard data from the back-end storage component.

`subscribe(Query query)`

The `subscribe` method allows a *StorageModule* to specify its interest in specific captured or transduced information. For example, a *Repository* can subscribe for pictures taken in Atlanta or San Francisco. The *StorageModule* sends to the *Registry* a `DATA_SUBSCRIPTION` message containing the *Query* encoded as a Base-64 ASCII string. The

Registry uses this subscription to determine whether to relay `CAPTURED_DATA`, `UPDATED_DATA`, or `TRANSDUCED_DATA` messages to a *StorageModule.* If a *StorageModule* has not subscribed for any specific content, then the *Registry* relays all captured, updated or transduced information to it.

```
stateChanged(boolean state)
```

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.4 The AccessModule

The *AccessModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to obtain information from other parts of the system. The *AccessModule* object has the following programming interface:

```
AccessModule(String host)
AccessModule(String host, int port_1)
AccessModule(String host, int port_1, int port_2)
```

Developers can instantiate an *AccessModule* object using any of

the three constructor methods above. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *AccessModule* object should connect. If only `port_1` is provided, the *AccessModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, the *AccessModule* object attempts to connect with the default port values (6789 and 6790).

`subscribe(Query query)`

The `subscribe` method allows an *AccessModule* to specify its interest in specific captured or transduced information. For example, an access interface can use this method to subscribe for pictures taken in Atlanta or San Francisco. The *AccessModule* sends to the *Registry* a `DATA_SUBSCRIPTION` message containing the *Query* encoded as a Base-64 ASCII string. The *Registry* uses this subscription to determine whether to relay `CAPTURED_DATA`, `UPDATED_DATA`, or `TRANSDUCED_DATA` messages to the *AccessModule.* If an *AccessModule* has not subscribed for any specific content, the *Registry* does not relay captured, updated or transduced information to it.

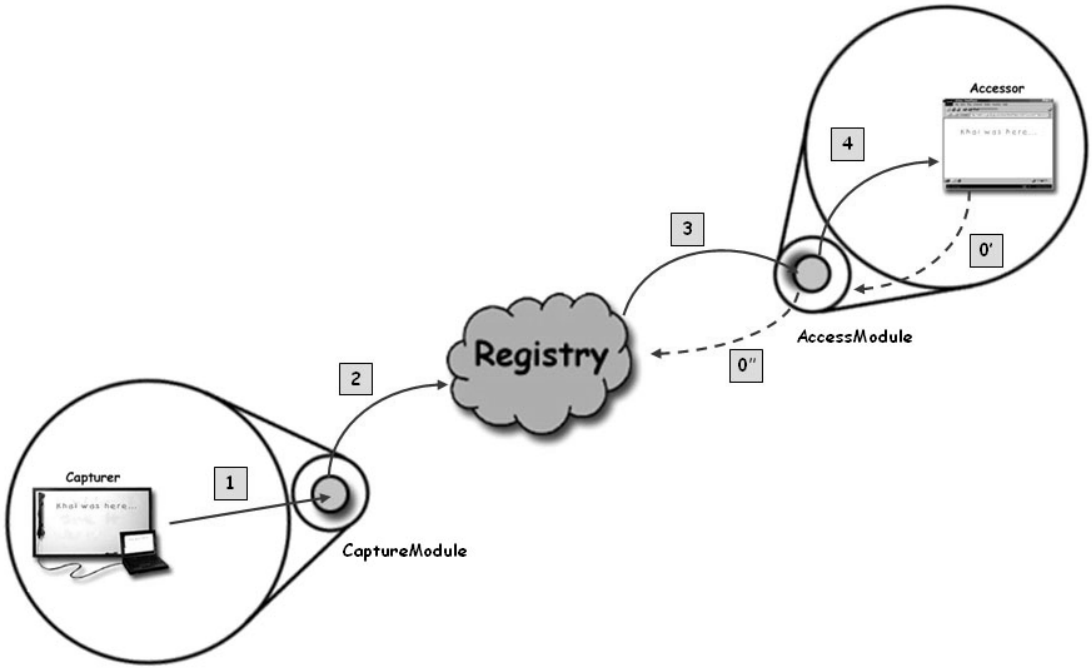


Figure A-4: Subscribing for information. (0') The access interface invokes the *AccessModule*'s `subscribe` method with a *Query* describing its interest in captured content. (0") The *AccessModule* sends to the *Registry* a `DATA_SUBSCRIPTION` message that includes the *Query* encoded as a Base-64 ASCII string. The *Registry* stores this subscription. (1 & 2) When an application captures information, the *Registry* uses determines if an access client has subscribed for that content and (3) relays to it the `CAPTURED_DATA` message. (4) The *AccessModule* invokes its `handle` callback method with the captured data converted back into a *DataObject*.

`handle(DataObject data)`

When the *AccessModule* receives a `CAPTURED_DATA`, `UPDATED_DATA`, or `TRANSDUCED_DATA` message, which contains a *DataObject* encoded as a Base-64 ASCII string, it invokes this callback method with that information converted back into a *DataObject* as a parameter.

`DataVector request(Query query)`

The `request` method allows a client application to retrieve stored information. When an application invokes this method, the *AccessModule* traverses the list of registered *Querier* objects to gather a list of additional query parameters. Once this process completes, the *AccessModule* sends to the *Registry* a `DATA_REQUESTED` message containing the full *Query* encoded as a Base-64 ASCII string. The *Registry* relays the message to *StorageModule* objects that may have the requested content. The *AccessModule* busy-waits until it receives from the *Registry* a `STORED_DATA` message, which contains a *DataVector* encoded as a Base-64 ASCII string. This method then returns the *DataVector* to the access application. See Figure A-3.

`AttributeVector getRelatedStoredAttributes(Query query)`

The `getRelatedStoredAttributes` method allows a client application to determine the list of attributes for the stored information that matches the specified *Query*. The *AccessModule* sends to the *Registry* a `RELATED_ATTRIBUTES_QUERY` message containing the *Query* encoded as a Base-64 ASCII string. The *Registry* relays the message to *StorageModule* while the *AccessModule* busy-waits until it receives from

the *Registry* a `RELATED_ATTRIBUTES_RESULT` message, which contains an *AttributeVector* encoded as a Base-64 ASCII string. This method then returns the *AttributeVector* to the access application.

`Vector getStoredAttributesClusters(String attribute_type)`

The `getRelatedStoredAttributes` method allows a client application to find the related attributes for all possible values of the specified attribute type. The *AccessModule* sends to the *Registry* an `ATTRIBUTES_CLUSTERS_QUERY` message containing the attribute type *String* encoded as a Base-64 ASCII string. The *Registry* relays the message to *StorageModule* while the *AccessModule* busy-waits until it receives from the *Registry* an `ATTRIBUTES_CLUSTERS_RESULT` message, which contains a *Vector* of *AttributeVectors* encoded as a Base-64 ASCII string. This method then returns the *Vector* to the access application.

`Vector getStoredAttributeTypes()`

The `getStoredAttributeTypes` method allows a client application to determine the list of attribute types used to describe the stored content. The *AccessModule* sends to the *Registry* an `ATTRIBUTE_TYPES_REQUESTED`. The *Registry* relays the message to *StorageModule* while the *AccessModule* busy-waits until it receives from

the *Registry* an `ATTRIBUTES_TYPES_RESULT` message, which contains a *Vector* of attribute type strings encoded as a Base-64 ASCII string. This method then returns the *Vector* to the access application.

`Vector getStoredAttributeValues(String type)`

The `getRelatedStoredAttributes` method allows a client application to get the list of the attribute values associated to a given attribute type. The *AccessModule* sends to the *Registry* an `ATTRIBUTE_VALUES_REQUESTED` message containing the attribute type *String* encoded as a Base-64 ASCII string. The *Registry* relays the message to *StorageModule* while the *AccessModule* busy-waits until it receives from the *Registry* an `ATTRIBUTE_VALUES_RESULT` message, which contains a *Vector* of attribute value strings encoded as a Base-64 ASCII string. This method then returns the *Vector* to the access application.

`addQuerier(Querier querier)`

The `addQuerier` method allows a client application to register a *Querier* object that will specify the information to retrieve.

`removeQuerier(Querier querier)`

The `removeQuerier` method allows a client application to unregister a *Querier* object.

`stateChanged(boolean state)`

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.5 The TransductionModule

The *TransductionModule* object extends the *Connection* object described in Appendix A.1.2. When there are mismatches between information used by access and storage components with what is captured, application developers create custom transducers in their applications by defining the transduce method to transform and transcode information between different data types (such as from a video file to a series of image frames) and formats (such as from a WAV file to an MP3 file). Overall, the *TransductionModule* object has the following programming interface:

```
TransductionModule(String host)
TransductionModule(String host, int port_1)
TransductionModule(String host, int port_1, int port_2)
```

Developers can instantiate a *TransductionModule* object using any of the three constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *TransductionModule* object should connect. If only `port_1` is provided, the *TransductionModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, the *TransductionModule* object attempts to connect with the default port values (6789 and 6790).

```
DataObject transduce(DataObject data)
```

When the *TransductionModule* object receives a `CAPTURED_DATA` or `UPDATED_DATA` message, which contains a *DataObject* encoded as a Base-64 ASCII string, it invokes this callback method with that information converted back into a *DataObject* as the parameter. When application developers extend the *TransductionModule* object, they must define this method to transform or transcode the information. The *TransductionModule* sends to the *Registry* a `TRANSDUCED_DATA` message containing the *DataObject* encoded as a Base-64 ASCII string. After the *Registry* has read the message, it relays

the message to the *AccessModule* or *StorageModule* objects that have an interest in the information.

`subscribe(Query query)`

The `subscribe` method allows a *TransductionModule* to specify its interest in specific captured information. For example, a transducer can use this method to subscribe for video that it converts to a sequence of images. The *TransductionModule* sends to the *Registry* a `DATA_SUBSCRIPTION` message containing the *Query* encoded as a Base-64 ASCII string. The *Registry* uses this subscription to determine whether to relay `CAPTURED_DATA` or `UPDATED_DATA` messages to the *TransductionModule.* If a *TransductionModule* has not subscribed for any specific content, then the *Registry* does not relay captured or updated information to it.

`stateChanged(boolean state)`

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.6 The ObserveModule

The *ObserveModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to learn about the run-time state of the system. The *TransductionModule* object has the following programming interface:

```
ObserveModule(String host)
ObserveModule(String host, int port_1)
ObserveModule(String host, int port_1, int port_2)
```

Developers can instantiate an *ObserveModule* object using any of the three constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *ObserveModule* object should connect. If only `port_1` is provided, the *ObserveModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, then the *ObserveModule* object attempts to connect with the default port values (6789 and 6790).

```
Vector listModules()
```

The `listModules` method allows an application to gain a detailed description of the run-time state of the system. The *ObserveModule* sends to the *Registry* a `MODULE_LISTING_REQUEST` message. The *ObserveModule* busy-waits while the *Registry* generates a

Vector of *ModuleDescriptor* objects, each describing a client module connected to the *Registry*. The *Registry* then sends back a `MODULE_LISTING_RESULT` message that contains the list encoded as a Base-64 ASCII string. This method then returns the *Vector* to the application.

`moduleAdded(ModuleDescriptor module)`

When the *Registry* connects with a new client module, the *Registry* sends to connected *ObserveModule* objects a `MODULE_ADDED` message, which contains a *ModuleDescriptor* for the new client module encoded as a Base-64 ASCII string. After the *ObserveModule* receives this message, it invokes this callback method with that information converted back into a *ModuleDescriptor* as the parameter.

`moduleRemoved(ModuleDescriptor module)`

When the *Registry* disconnects with a client module, the *Registry* sends to connected *ObserveModule* objects a `MODULE_REMOVED` message, which contains a *ModuleDescriptor* for the disconnected client module encoded as a Base-64 ASCII string. After the *ObserveModule* receives this message, it invokes this callback method with that information converted back into a *ModuleDescriptor* as the parameter.

`behaviorChanged(ModuleDescriptor module)`

When a client module changes its subscription, the *Registry* sends to connected *ObserveModule* objects a `BEHAVIOR_CHANGED` message, which contains a *ModuleDescriptor* for that client module encoded as a Base-64 ASCII string. After the *ObserveModule* receives this message, it invokes this callback method with that information converted back into a *ModuleDescriptor* as the parameter.

`stateChanged(boolean state)`

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.7 The ControlModule

The *ControlModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to update the run-time state of the system. This component must be used in conjunction with the *ObserveModule*. The `control` function requires a *ModuleDescriptor* parameter. We do not allow a developer to instantiate the *ModuleDescriptor* object class. Instead, a

specific *ModuleDescriptor* handle can be obtained as part of the list of descriptors returned by the *ObserveModule*'s `listModules` function or the `stateChanged` callback function. The *ControlModule* object has the following programming interface:

```
ControlModule(String host)
ControlModule(String host, int port_1)
ControlModule(String host, int port_1, int port_2)
```

Developers can instantiate a *ControlModule* object using any of the three constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *ControlModule* object should connect. If only `port_1` is provided, the *ControlModule* object automatically attempts to connect with the next port number as the second port value. If no port values are specified, the *ControlModule* object attempts to connect with the default port values (6789 and 6790).

```
control(ModuleDescriptor module)
```

The `control` method allows an application to update the state of a client module in the system. The *ControlModule* sends to the *Registry* a `CHANGE_STATE` message, which contains a *ModuleDescriptor* for the client module to update encoded as a Base-64 ASCII string. After the *Registry* reads the message and identifies the corresponding client module, it relays the `CHANGE_STATE` message to that module.

```
stateChanged(boolean state)
```

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

A.2.8 The GarbageCollectionModule

The *GarbageCollectionModule* object extends the *Connection* object described in Appendix A.1.2. Application developers can use this object in their applications to perform the attribute-triggered discarding of unwanted information. The *GarbageCollectionModule* object has the following programming interface:

```
GarbageCollectionModule(String host)
GarbageCollectionModule(String host, int port_1)
GarbageCollectionModule(String host, int port_1, int port_2)
```

Developers can instantiate a *GarbageCollectionModule* object using any of the three constructor methods. The developer can specify the host name or IP address of the server and up to two port numbers of the *Registry* object with which the *GarbageCollectionModule* object should connect. If only `port_1` is provided, the *GarbageCollectionModule* object automatically attempts to connect with the next port number as the

second port value. If no port values are specified, the *GarbageCollectionModule* object attempts to connect with the default port values (6789 and 6790).

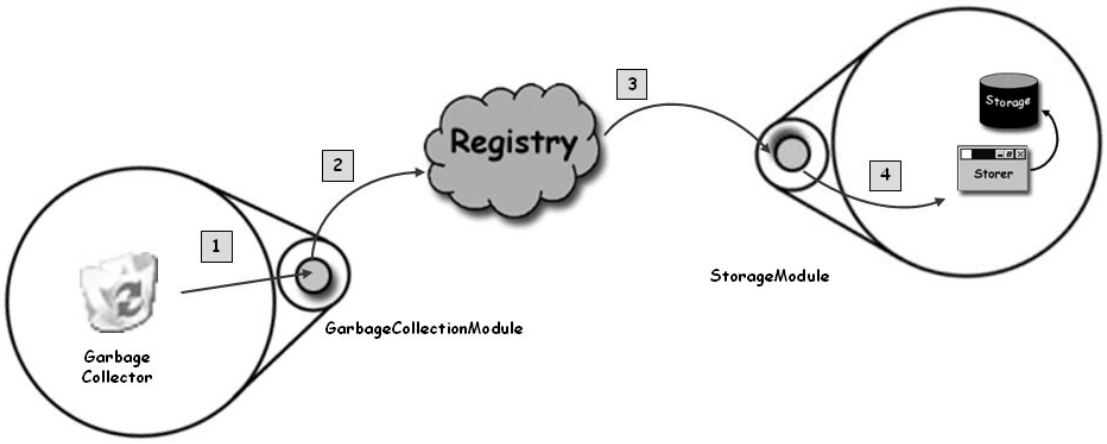


Figure A-5: Discarding content. (1) The garbage collector invokes the *GarbageCollectionModule*'s `gc` method with a *Query* describing the stored content to discard. (2) The *GarbageCollectionModule* sends to the *Registry* a `GARBAGE_COLLECT` message that includes the *Query* encoded as a Base-64 ASCII string. (3) The *Registry* relays this message to all storage modules. (4) The *StorageModule* invokes its `gc` callback method with the discard request converted back into a *Query*.

`gc(Query query)`

Similar to how information desired for review can be specified using a query over the attribute tags, an application discards content by invoking the *GarbageCollectionModule*'s `gc` function with a *Query* specifying attributes of data she wants storage repositories to remove from persistence. The *GarbageCollectionModule* sends to the *Registry* a `GARBAGE_COLLECT` message, which contains a *Query* specifying data to discard encoded as a Base-64 ASCII string. After the *Registry* reads the

message, it relays the message to all storage modules. In the current implementation, storage repositories actually delete the data, meaning that the discarded content can never be recovered after that point.

`stateChanged(boolean state)`

When the module successfully connects with the *Registry,* it invokes this callback method with a true boolean parameter. When the client module loses its connection with the *Registry*, it invokes this callback method with a false boolean parameter, allowing applications to react accordingly based on the client's connectivity.

APPENDIX B

SAMPLE IMPLEMENTATION OF TAGGER OBJECT

We provide in this appendix the full source code for the *PeoplePresentTagger* object discussed in Section 4.2.1. This *Tagger* object associates captured data with the names of people present in a specified location. We highlight the minimal lines of code needed to extend the *Tagger* object with custom behavior. The remaining lines of code generate the specific attributes to be associated to the captured content.

```
import edu.gatech.coc.inca.arch.constants.*;
import edu.gatech.coc.inca.arch.data.*;
import edu.gatech.coc.inca.arch.format.*;

import edu.gatech.coc.inca.arch.tagger.*;

import java.util.*;

//// import objects from the Context Toolkit library that will be used...
import context.arch.comm.DataObject;
import context.arch.storage.AttributeNameValue;
import context.arch.storage.AttributeNameValues;

/**
 * Object that generates attribute tags specifying people present in a location
 */
public class PeoplePresentTagger
  implements Tagger
{
  //-------------------------------------------------------------------------
  // STATIC VARIABLE(S)
  //-------------------------------------------------------------------------

  public static final int DEFAULT_SIGNPOST_SUBSCRIBER_PORT = 9800;

  //-------------------------------------------------------------------------
  // INSTANCE VARIABLE(S)
  //-------------------------------------------------------------------------

  protected String location; // name of the location to tag for
  protected Vector people_present = new Vector(); // people present in location

  //-------------------------------------------------------------------------
  // CONSTRUCTOR(S)
  //-------------------------------------------------------------------------

  /**
   * Constructor.
   * @param location is the String value name of the location to monitor for
   *   people present in
   */
  public PeoplePresentTagger(String location)
  {
    this.location = location;

    //// creates a context toolkit subscriber component for the sign post
    //// application the signpost application monitors context for all the
```

```
    //// rooms in a home
    SignPostSubscriber signpost_subscriber =
      new SignPostSubscriber(DEFAULT_SIGNPOST_SUBSCRIBER_PORT) {

        /**
         * Override its handle method.  It is called when the callbacks this
         * object subscribes to are triggered.  So, when there's new data from
         * the signpost widget, this is run.
         */
        public DataObject handle(String callback,
                                 context.arch.comm.DataObject data)
          throws InvalidMethodException, MethodException
        {
          AttributeNameValues atts = new AttributeNameValues(data);
          AttributeNameValue user = atts.getAttributeNameValue(USERNAME);
          AttributeNameValue new_location =
            atts.getAttributeNameValue(ContextTypes.LOCATION);
          AttributeNameValue time =
            atts.getAttributeNameValue(Widget.TIMESTAMP);

          if (location.equals(new_location))
          {
            if (people_present.indexOf(user) == -1)
              people_present.addElement(user);
          }
          else
          {
            if (people_present.indexOf(user) != -1)
              people_present.removeElement(user);
          }

          return null;
        }
      };

  } // end of PeoplePresentTagger() constructor

  //-------------------------------------------------------------------------
  // INSTANCE METHOD(S)
  //-------------------------------------------------------------------------

  /**
   * Gets an AttributeVector of Attributes to tag to the DataObject specified.
   * This is a method that all objects implementing the Tagger interface must
   * defined.
   * @param data_object is the DataObject to tag attributes to
   * @return the vector of new attributes to add to the data object.
   */
  public AttributeVector getAttributes(DataObject data_object)
  {
    AttributeVector new attributes = new AttributeVector();
    new attributes.addElement(
       new Attribute(AttributeConstants.ATTRIBUTE_NAME__LOCATION,location));
    for (int i=0; i<people_present.size(); i++)
    {
      String person = (String) people_present.elementAt(i);
      new attributes.addElement(
        new Attribute(AttributeConstants.ATTRIBUTE_NAME__PERSON,person));
    }

    return(new_attributes);
  } // end of getAttributes()

} // end of PeoplePresentTagger class
```

APPENDIX C

SAMPLE IMPLEMENTATION OF QUERIER OBJECT

We provide in this appendix the full source code for the *TimeQuerier* object discussed in Section 4.2.3. In this example, we demonstrate that a *Querier* object can be an interactive GUI component that exists as part of the access interface. This *TimeQuerier* object acquires a user defined query, where the user inputs time parameters specifying what information to retrieve. We highlight the minimal lines of code needed to extend the *Querier* object with custom behavior. The remaining lines of code create the GUI for the *TimeQuerier* component shown in Figure 4-3. Depending on how the developer intends to obtain the time parameters, the number of additional lines of code, such as those shown for the GUI, will differ.

```
import java.util.*;
import java.awt.*;
import javax.swing.*;

import edu.gatech.coc.inca.arch.data.*;
import edu.gatech.coc.inca.arch.constants.*;
import edu.gatech.coc.inca.arch.querier.*;

/**
 * Component that allows the user to specify time parameters of the information
 *   to retrieve.
 */
class TimeQuerier
  extends JPanel
  implements Querier
{
  //---------------------------------------------------------------------------
  // STATIC VARIABLE(S)
  //---------------------------------------------------------------------------

  //// options for months,days, years, hours and minutes users can pick from
  protected static String[] months = {"January","February","March","April",
                                      "May","June","July","August","September",
                                      "October","November","December"};
  protected static String[] days;
  protected static String[] years;
  protected static String[] hours;
  protected static String[] minutes;

  //---------------------------------------------------------------------------
  // INSTANCE VARIABLE(S)
  //---------------------------------------------------------------------------
```

```
//// select (drop down boxes) for selecting the start and stop time
////  where time = month, day, year, hour and minute
protected JComboBox start_month, start_day, start_year, start_hour,
                    start_minute;
protected JComboBox stop_month, stop_day, stop_year, stop_hour, stop_minute;

//--------------------------------------------------------------------------
// CONSTRUCTOR(S)
//--------------------------------------------------------------------------

/**
 * Static constructor, used to initialize all the variables
 */
static
{
  days = new String[31];
  for (int i=0; i<days.length; i++)
    days[i] = new Integer(i+1).toString();

  //// determine the first year possible (epoch)
  Calendar calendar = Calendar.getInstance();
  calendar.setTime(new Date(0));
  int first_year = calendar.get(Calendar.YEAR);
  calendar.setTime(new Date(System.currentTimeMillis()));
  //// determine the last year possible (current year)
  int current_year = calendar.get(Calendar.YEAR);

  //// populate the list of possible years
  years = new String[current_year-first_year+1];
  for (int i=0; i<years.length; i++)
    years[i] = new Integer(first_year+i).toString();

  //// populate the list of possible hours (0-23)
  hours = new String[24];
  for (int i=0; i<hours.length; i++)
  {
    String str = new Integer(i).toString();
    while (str.length() < 2) // force it into being a 2 digit string
      str = "0" + str;
    hours[i] = str;
  }

  //// populate the list of possible minutes (0-59)
  minutes = new String[60];
  for (int i=0; i<minutes.length; i++)
  {
    String str = new Integer(i).toString();
    while (str.length() < 2) // force it into being a 2 digit string
      str = "0" + str;
    minutes[i] = str;
  }
} // end of static constructor

/**
 * Instance constructor
 */
public TimeQuerier()
{
  //// creates the start and stop time panels
  JPanel start_panel = createTimePanel("Start: ",
                                start_year = new JComboBox(years),
                                start_month = new JComboBox(months),
                                start_day = new JComboBox(days),
                                start_hour = new JComboBox(hours),
                                start_minute = new JComboBox(minutes));

  JPanel stop_panel = createTimePanel("Stop:  ",
                               stop_year = new JComboBox(years),
                               stop_month = new JComboBox(months),
                               stop_day = new JComboBox(days),
                               stop_hour = new JComboBox(hours),
                               stop_minute = new JComboBox(minutes));

  //// add them to the main GUI panel
  setLayout(new GridLayout(2,1));
  add(start_panel);
  add(stop_panel);
} // end of TimeQuerier() constructor
```

```
  //---------------------------------------------------------------------------
  // INSTANCE METHOD(S)
  //---------------------------------------------------------------------------

  /**
   * Creates the GUI panel for specifying date/time
   * @param label is the String label for this panel (usually start or stop)
   * @param year is the drop down combo box for selecting the year in the date
   * @param month is the drop down combo box for selecting the month in the date
   * @param day is the drop down combo box for selecting the day in the date
   * @param hour is the drop down combo box for selecting the hour in the time
   * @param minute is the drop down combo box for selecting the minute
   */
  protected JPanel createTimePanel(String label, JComboBox year,
                                   JComboBox month, JComboBox day,
                                   JComboBox hour, JComboBox minute)
  {
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());

    panel.add(new JLabel(label));
    panel.add(month);
    panel.add(day);
    panel.add(new JLabel(","));
    panel.add(year);
    panel.add(new JLabel("  "));
    panel.add(hour);
    panel.add(new JLabel(":"));
    panel.add(minute);

    return(panel);
  } // end of createTimePanel()

  /**
   * Obtain the time query from the GUI
   * @return a Query object
   */
  public Query getQuery()
  {
    //// grab the start & stop time from the GUI
    long start_time = getTime(start_month,start_day,start_year,start_hour,
                              start_minute);
    long stop_time = getTime(stop_month,stop_day,stop_year,stop_hour,
                             stop_minute);

    //// create a query for objects with start_time <= timestamp AND
   /////   timestamp <= stop_time
    Query q = new Query();
    q.greaterThanEquals(new Attribute(
                            AttributeConstants.ATTRIBUTE NAME _TIMESTAMP,
                            new Long(start time).toString()));
    q.lessThanEquals(new Attribute(AttributeConstants.ATTRIBUTE NAME__TIMESTAMP,
                                   new Long(stop_time).toString()));

    return(q);
  } // end of getQuery()

  /**
   * Obtain the time from the GUI components
   * @param month is the GUI drop down combobox for selecting the month
   * @param day is the GUI drop down combobox for selecting the day
   * @param year is the GUI drop down combobox for selecting the year
   * @param hour is the GUI drop down combobox for selecting the hour
   * @param minute is the GUI drop down combobox for selecting the minute
   * @return long time value
   */
  protected long getTime(JComboBox month, JComboBox day, JComboBox year,
                         JComboBox hour, JComboBox minute)
  {
    Calendar calendar = Calendar.getInstance();
    calendar.clear();
    calendar.set(Integer.parseInt((String)year.getSelectedItem()),
                 Integer.parseInt((String)month.getSelectedItem()),
                 Integer.parseInt((String)day.getSelectedItem()),
                 Integer.parseInt((String)hour.getSelectedItem()),
                 Integer.parseInt((String)minute.getSelectedItem()));
    return(calendar.getTime().getTime());
  } // end of getTime()
} // end of TimeQuerier class
```

APPENDIX D

PERSONAL AUDIO LOOP SOURCE CODE

We provide in this appendix the full source code for the Personal Audio Loop application discussed in Chapter 5. We divide this capture and access problem into four smaller concerns. The main method instantiates these four parts as well as a *Registry* object that integrates them. Overall, the source code below contains 307 lines of code and comments. Of the 158 lines of actual code, we use 51 lines to access, capture, store and discard captured information. We show that the capture and storage concerns can be addressed through reusable components available in the INCA toolkit. We illustrate how to construct an access interface and how to customize the garbage collection behavior.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

import edu.gatech.coc.inca.arch.data.*;
import edu.gatech.coc.inca.arch.format.*;
import edu.gatech.coc.inca.arch.module.*;


/**
 * Personal Audio Loop application.
 */
class PAL
  extends Frame
  implements ActionListener,
             WindowListener,
             AudioPlayerListener
{
  //============================================================================
  // INSTANCE VARIABLE(S)
  //============================================================================

  // audio player component for requesting audio
  // it automatically plays the audio
  protected AudioPlayer audio_player = null;

  protected long playback_time = -1;

  protected Button nudge_left = new Button("<<");
  protected Button nudge_right = new Button(">>");
  protected Button stop = new Button("Stop");
```

```
protected Label datetime = new Label(" ",Label.CENTER);

//============================================================================
// CONSTRUCTOR(S)
//============================================================================

/**
 * Default Constructor.
 */
public PAL()
{
  this("localhost");
}

/**
 * Default Constructor.
 */
public PAL(String host)
{
  this.setIconImage(getToolkit().getImage("images/new_icon2.gif"));
  this.setTitle("pAL");
  this.setBackground(Color.black);
  this.addWindowListener(this);

  datetime.setFont(new Font("Dialog",Font.PLAIN,10));
  datetime.setBackground(Color.black);
  datetime.setForeground(Color.green);

  Panel panel = new Panel();
  panel.setLayout(new FlowLayout());
  panel.setBackground(Color.black);
  panel.add(nudge_left);
  panel.add(nudge_right);
  panel.add(new Label("  "));
  panel.add(stop);
  nudge_left.addActionListener(this);
  nudge_right.addActionListener(this);
  stop.addActionListener(this);

  setBackground(Color.lightGray);
  setLayout(new BorderLayout());
  add(BorderLayout.NORTH,datetime);
  add(BorderLayout.SOUTH,panel);

  setBackground(Color.black);
  setSize(160,80);
  setVisible(true);
  setResizable(false);

  // audio player component for requesting audio
  // it automatically plays the audio
  audio_player = new AudioPlayer();

  // subscribes a listener to the player
  audio_player.addListener(this);

} // end of PAL()

//============================================================================
// INSTANCE METHOD(S)
//============================================================================

/**
 * Callback method for actionlistener
 * @param e is the ActionEvent to handle
 */
public void actionPerformed(ActionEvent e)
{
  Object source = e.getSource();

  if (source == stop)
  {
    datetime.setText(new String());
    playback_time = -1;

    stopPlayback();
  }
```

```
  else if (source == nudge_left)
  {
    stopPlayback();

    if (playback_time == -1)
      playback_time = System.currentTimeMillis();

    playback_time -= (30 * 1000); // move back in time by 30 seconds
    playback();
  }
  else if (source == nudge_right)
  {
    stopPlayback();

    if (playback_time == -1)
      playback_time = System.currentTimeMillis();
    playback_time += (7 * 1000); // move forward in time by 7 seconds

    playback();
  }
} // end of actionPerformed()

/**
 * Stops the playback of audio
 */
protected void stopPlayback()
{
  audio_player.stopPlayback();
  datetime.setText(" ");
} // end of stopPlayback()

/**
 * Starts the playback of audio
 */
protected void playback()
{
  // create a query for information starting from time t until
  // (t + 60 seconds)
  Query q_start_time = new Query();
  q_start_time.greaterThan(new Attribute("TimeStamp",
    new Long(playback_time).toString()));
  Query q_stop_time = new Query();
  q_stop_time.lessThan(new Attribute("TimeStamp",
    new Long(t+(1*60*1000)).toString()));
  Query q_main = new Query();
  q_main.and(q_start_time);
  q_main.and(q_stop_time);

  // use the audio_player to request the audio (and automatically playback
  // the audio)
  audio_player.playback(q_main);
} // end of playback()

/**
 * Callback method for the AudioPlayerListener to be informed of the playback
 *   time
 * @param time is the current playback time
 */
public void isPlaying(long time)
{
  this.playback_time = time;

  Date d = new Date(time);
  String strTime = toTwoDigits(d.getHours()) + ":" +
    toTwoDigits(d.getMinutes()) + ":" + toTwoDigits(d.getSeconds());
  datetime.setText(strTime);
} // end of isPlaying()

/**
 * Converts an int value into a string of at least two digits (starting with
 *   '0'if necessary
 * @param i is the int
 * @return String equivalent of the int value
 */
public String toTwoDigits(int i)
{
  String str = new Integer(i).toString();
  while(str.length() < 2)
    str = "0" + str;
```

```
    return(str);
  } // end of toTwoDigits()

  /**
   * Callback method for WindowListener when window is closing.
   * @param e is the WindowEvent to handle.
   */
  public void windowClosing(WindowEvent e)
  {
    Window window = e.getWindow();
    window.setVisible(false);
    window.dispose();

    System.exit(0);
  } // end of windowClosing()

  public void windowClosed(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}


  //============================================================================
  // MAIN METHOD
  //============================================================================

  /**
   * Main method.
   * @param argv is the String[] command line argument.
   */
  public static void main(String argv[])
  {
    //// start the registry...
    InCAd registry = new InCAd();

    //// start the repository
    Repository repository = new Repository();
    //// start garbage collector component that will periodically discard audio
    PeriodicGarbageCollector gc = new PeriodicGarbageCollector();

    //// starts the capture application
    WaveCapturer wave_capturer = new WaveCapturer("localhost");
    wave_capturer.addTagger(new TimeStampTagger());
    wave_capturer.startCapture();

    //// starts the access application
    PAL pal = new PAL();

  } // end of main()

} // class PAL



import edu.gatech.coc.inca.arch.data.*;
import edu.gatech.coc.inca.arch.format.*;
import edu.gatech.coc.inca.arch.module.*;

/**
 * Class for periodically discarding content older than a fixed amount of time
 *   in this case: 15 minutes.
 */
class PeriodicGarbageCollector
  extends GarbageCollectionModule
  implements Runnable
{
  //============================================================================
  // INSTANCE VARIABLE(S)
  //============================================================================

  //// thread object
  protected Thread thread;
```

```
  /**
   * Constructor
   */
  public PeriodicGarbageCollector()
  {
    thread = new Thread(this);
    thread.start();
  } // end of PeriodicGarbageCollector() constructor

  //============================================================================
  // INSTANCE METHOD(S)
  //============================================================================

  /**
   * Run loop for the thread
   */
  public void run()
  {
    while(true)
    {
      try
      {
        Thread.sleep(1000 * 60);
        Query q_time = new Query();
        q_time.lessThan(new Attribute("TimeStamp",
          new Long(System.currentTimeMillis() - (15 * 60* 1000)).toString()));
        gc(q_time);  // remove data older than a certain time from storage
      }
      catch(Exception e)
      {
      }
    }
  } // end of run()

} // end of PeriodicGarbageCollector class
```

REFERENCES

1. Abowd, G.D. Classroom 2000: An Experiment with the Instrumentation of a Living Educational Environment. *IBM Systems Journal, **38**(4):* pp.508-530 (1999)

2. Abowd, G.D. Software Engineering Issues for Ubiquitous Computing. In the *Proceedings of the 1999 International Conference on Software Engineering:* pp.75-84, Los Angeles, California, USA, ACM Press (May 16-22, 1999)

3. Abowd, G.D., C.G. Atkeson, J. Brotherton, T. Enqvist, P. Gulley and J. LeMon. Investigating the Capture, Integration and Access Problem of Ubiquitous Computing in an Educational Setting. In the *Proceedings of the CHI 98 Conference on Human Factors in Computing Systems:* pp. 440-447, Los Angeles, California, USA, ACM Press (April 18-23, 1998)

4. Abowd, G.D., C.G. Atkeson, A. Feinstein, C. Hmelo, R. Kooper, S. Long, N. Sawhney and M. Tani. Teaching and Learning as Multimedia Authoring: The Classroom 2000 Project. In the *Proceedings of the Fourth ACM International Conference on Multimedia'96:* pp. 187-198, Boston, Massachusetts, USA, ACM Press (November 18-22, 1996)

5. Abowd, G.D., M. Gauger and A. Lachenmann. The Family Video Archive: An Annotation and Browsing Environment for Home Movies. In the *Proceedings of the 5th ACM SIGMM International Workshop on Multimedia Information Retrieval:* pp. 1-8, Berkeley, California, USA, ACM Press (November 7, 2003)

6. Abowd, G.D. and E.D. Mynatt. Charting Past, Present, and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction, **7**(1):* pp. 29-58 (2000)

7. Bacher, C., R. Müller, T. Ottmann and M. Will. Authoring on the Fly: A New Way of Integrating Telepresentation and Courseware Production. In the *Proceedings of the 5th International Conference on Computers in Education:* Kuching, Sarawak, Malaysia (December 1997)

8. Beckmann, C., S. Consolvo and A. LaMarca. Some Assembly Required: Supporting End-User Sensor Installation in Domestic Ubiquitous Computing Environments. In the *Proceedings of the 6th International Conference on Ubiquitous Computing:* pp. 107-124, Nottingham, UK, Springer-Verlag (September 7-10, 2004)

9. Bell, G. A Personal Digital Store. *Communications of the ACM, **44**(1):* pp. 86-

91 (2001)

10. Berque, D., A. Hutcheson, D. Johnson, L. Jovanovic, K. Moore, C. Singer and K. Slattery. Using a Variation of the WYSIWIS Shared Drawing Surface Paradigm to Support Electronic Classrooms. Short paper & poster presented at the *8th International Conference on Human Computer Interaction (HCI'99)*: Munich, Germany, IRB-Verlag (August 22-27, 1999)

11. Bianchi, M. AutoAuditorium: A Fully Automatic, Multi-Camera System to Televise Auditorium Presentations. Paper presented at the *Joint DARPA/NIST Smart Spaces Workshop:* Gaithersburg, Maryland (July 30-31, 1998)

12. Black, M., F. Berard, A. Jepson, W. Newman, E. Saund, G. Socher and M. Taylor. Digital Office: Overview. In the *Proceedings of the AAAI Spring Symposium on Intelligent Environments:* pp. 1-6, Stanford, California, USA (March 23-25, 1998)

13. Brotherton, J.A. Enriching Everyday Activities Through the Automated Capture and Access of Live Experiences—eClass: Building, Observing and Understanding the Impact of Capture and Access in an Educational Domain. Ph.D. Dissertation: 270 pages, Georgia Institute of Technology, Atlanta, Georgia, USA (2001)

14. Brotherton, J.A. and G.D. Abowd. Lessons Learned from eClass: Assessing Automated Capture and Access in the Classroom. *ACM Transactions on Computer-Human Interaction, **11**(2):* pp. 121-155 (2004)

15. Brotherton, J.A., G.D. Abowd and K.N. Truong. Supporting Capture and Access Interfaces for Informal and Opportunistic Meetings. Technical Report GIT-GVU-99-06: Georgia Institute of Technology, Atlanta, Georgia, USA (1999)

16. Brotherton, J.A., J.R. Bhalodia and G.D. Abowd. Automated Capture, Integration, and Visualization of Multiple Media Streams, In the *Proceedings of the IEEE International Conference on Multimedia Computing and Systems:* pp. 54-63, Austin, Texas, USA, IEEE Computer Society Press (June 7-11, 1998)

17. Bush, V. As We May Think. *The Atlantic Monthly, **176**(1):* pp. 101-108 (1945)

18. Buxton, W. and T.P. Moran. EuroPARC's Integrated Interactive Intermedia Facility (IIIF): Early Experiences. In the *Proceedings of the IFIP WG8.4 Conference on Multi-User Interfaces and Applications:* pp. 11-34, Heraklion, Crete, Greece (September 24-26, 1990)

19. Chiu, P., A. Kapuskar, S. Reitmeier and L. Wilcox. NoteLook: Taking Notes in Meetings with Digital Video and Ink. In the *Proceedings of the Seventh ACM*

International Conference on Multimedia: pp. 149-158, Orlando, Florida, USA, ACM Press (October 30-November 5, 1999)

20. Chiu, P. and K.N. Truong. Interactive Space-Time Planes for Document Visualization. In the *Compendium of IEEE Symposium on Information Visualization:* pp. 10-11, Boston, Massachusetts, USA (October 28-29, 2002)

21. Cruz, G. and R. Hill. Capturing and Playing Multimedia Events with STREAMS. In the *Proceedings of the Second ACM International Conference on Multimedia:* pp. 193-200, San Francisco, California, USA, ACM Press (October 15-20, 1994)

22. Davis, R.C., J.A. Landay, V. Chen, J. Huang, R.B. Lee, F.C. Li, J. Lin, Charles B. Morrey, III, B. Schleimer, M.N. Price, and B.N. Schilit. NotePals: Lightweight Note Sharing by the Group, For the Group. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 338-345, ACM Press: Pittsburgh, Pennsylvania, USA, ACM Press (May 15-20, 1999)

23. Dey, A.K. Providing Architectural Support for Building Context-Aware Applications. Ph.D. Dissertation: 240 pages, Georgia Institute of Technology, Atlanta, Georgia, USA (2000)

24. Dey, A.K., D. Salber, G.D. Abowd and M. Futakawa. The Conference Assistant: Combining Context-Awareness with Wearable Computing. In the *Proceedings of the 3rd IEEE International Symposium on Wearable Computers:* pp. 21-28, San Francisco, California, USA, IEEE Computer Society Press (October 18-19, 1999).

25. Dietz, P.H. and W.S. Yerazunis. Real-Time Audio Buffering for Telephone Applications. In the *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology:* pp. 193-194, Orlando, Florida, USA, ACM Press (November 11-14, 2001)

26. Dourish, P., A. Adler, V. Bellotti and A. Henderson. Your Place or Mine? Learning from Long-Term Use of Audio-Video Communication. *Journal of Computer-Supported Cooperative Work, **5**(1):* pp. 33-62 (1996)

27. Dourish, P., W.K. Edwards, A. LaMarca and M. Salisbury. Presto: an Experimental Architecture for Fluid Interactive Document Spaces. *ACM Transactions on Computer-Human Interaction, **6**(2):* pp. 133-161 (1996)

28. Eldridge, M., M. Lamming and M. Flynn. Does a video diary help recall? In the *Proceedings of the HCI'92 Conference on People and Computers VII:* pp. 257-269, York, UK, Cambridge University Press (August 15-18, 1992)

29. Elrod, S., R. Bruce, R. Gold, D. Goldberg, F. Halasz, W. Janssen, D. Lee, K. McCall, E. Pedersen, K. Pier, J. Tang, and B. Welch. Liveboard: A Large Interactive Display Supporting Group Meetings, Presentations, and Remote Collaboration. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 599-607, Monterey, California, USA, ACM Press (May 3-7, 1992)

30. Engelbart, D.C., Augmenting human intellect: A conceptual framework, Summary report. 1962, AFOSR-3223 - Contract AF49(638)-1024, SRI Project 3578: Air Force Office of Scientific Research.

31. Fleck, M., M. Frid, T. Kindberg, E. O'Brien-Strain, R. Rajani and M. Spasojevic. Rememberer: A Tool for Capturing Museum Visits. In the *Proceedings of the 4th International Conference on Ubiquitous Computing:* pp. 48-55, Göteborg, Sweden, Springer-Verlag (September 29-October 1, 2002)

32. Freeman, E.T. The Lifestreams Software Architecture. Ph.D. Dissertation: 185 pages, Yale University, New Haven, Connecticut, USA (1997)

33. Gemmell, J., G. Bell, R. Lueder, S. Drucker and C. Wong. MyLifeBits: Fulfilling the Memex Vision. In the *Proceedings of the Tenth ACM International Conference on Multimedia:* pp. 235-238, Juan-les-Pins, France, ACM Press (December 1-6, 2002).

34. Gemmell, J., L. Williams, K. Wood, R. Lueder and B. Gordon. Passive Capture and Ensuing Issues for a Personal Lifetime Store. In the *Proceedings of the 1st ACM Workshop on Continuous Archival and Retrieval of Personal Experiences:* pp. 48-55, New York, New York, USA, ACM Press (October 15, 2004)

35. Gleicher, M. L., R. M. Heck and M. N. Wallick. A Framework for Virtual Videography. In the *Proceedings of the 2nd International Symposium on Smart Graphics:* pp. 9-16, Hawethorne, New York, USA, ACM Press (June 11-13, 2002)

36. Greenberg, S. and C. Fitchett. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. In the *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology:* pp. 209-128, Orlando, Florida, USA, ACM Press (November 11-14, 2001)

37. Grudin, J. Groupware and Social Dynamics: Eight Challenges for Developers.

*Communications of the ACM, **37**(1):* pp. 92-105 (1994)

38. Guzdial, M., J. Rick and B. Kerimbaev. Recognizing and supporting roles in CSCW. In the *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work:* pp. 261-268, Philadelphia, Pennsylvania, USA, ACM Press (December 2-6, 2000)

39. Hâkansson, M., S. Ljungblad and L.E. Holmquist. Capturing the Invisible: Designing Context-Aware Photography. In the *Proceedings of the 2003 Conference on Designing for User Experiences:* pp. 1-4, San Francisco, California, USA, ACM Press (June 6-7, 2003)

40. Hayes, G.R., J.A. Kientz, K.N. Truong, D.R. White, G.D. Abowd and T. Pering. Designing Capture Applications to Support the Education of Children with Autism. In the *Proceedings of the 6th International Conference on Ubiquitous Computing:* pp. 161-178, Nottingham, UK, Springer-Verlag (September 7-10, 2004)

41. Hayes, G.R., S.N. Patel, K.N. Truong, G. Iachello, J.A. Kientz, R. Farmer and G.D. Abowd. The Personal Audio Loop: Designing a Ubiquitous Audio-Based Memory Aid. In the *Proceedings of the 6th International Symposium on Mobile Human-Computer Interaction:* pp. 168-179, Glasgow, UK, Springer-Verlag (September 13-16, 2004)

42. Hayes, G.R., J.S. Pierce and G.D. Abowd. User Trends in the Capture and Access of Short Important Thoughts. Technical Report GIT-GVU-03-09: Georgia Institute of Technology, Atlanta, Georgia, USA (2003)

43. Hayes, G.R., K.N. Truong, G.D. Abowd and T. Pering. Experience Buffers: A Socially Appropriate, Selective Archiving Tool for Evidence-Based Care. In the *Extended Abstracts of the Conference on Human Factors and Computing Systems:* pp. 1435-1438, Portland, Oregon, USA, ACM Press (April 2-9, 2005)

44. Healey, J. and R.W. Picard. StartleCam: A Cybernetic Wearable Camera. In the *Proceedings of the 2nd IEEE International Symposium on Wearable Computers:* pp. 42-49, Pittsburgh, Pennsylvania, USA, IEEE Computer Society Press (October 19-20, 1998)

45. Hindus, D. and C. Schmandt. Ubiquitous Audio: Capturing Spontaneous Collaboration. In the *Proceedings of the Conference on Computer-Supported Cooperative Work:* pp. 210-217, Toronto, Ontario, Canada, ACM Press (October 31-November 4, 1992)

46. Huang, J. A Collaborative Property-Based Note Management System. Master's Dissertation: 48 pages, University of California at Berkeley, Berkeley,

California, USA (2000)

47. Iachello, G. and G.D. Abowd. A Token-based Access Control Mechanism for Automated Capture and Access Systems in Ubiquitous Computing. Technical Report GIT-GVU-05-06: Georgia Institute of Technology, Atlanta, Georgia, USA (1999)

48. Janssen, B., ILU Manual. Xerox Technical Report ISTL-CSA-94-01-02, Xerox PARC, Palo Alto, California, USA (1994)

49. Ju, W., A. Ionescu, L. Neeley and T. Winograd. Where the Wild Things Work: Capturing Shared Physical Design Workspaces. In the Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work: pp. 533-541, Chicago, Illinois, USA, ACM Press (November 6-10, 2004)

50. Klemmer, S.R., J. Li, J. Lin and J.A. Landay. Papier-Mache: Toolkit Support for Tangible Input. In the *Proceedings of the 2004 Conference on Human Factors in Computing Systems:* pp. 399-406, Vienna, Austria, ACM Press (April 24-29, 2004)

51. Klemmer, S.R., M.W. Newman, R. Farrell, M. Bilezikjian and J.A. Landay. The Designers' Outpost: A Tangible Interface for Collaborative Web Site. In the *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology:* pp. 1-10, Orlando, Florida, USA, ACM Press (November 11-14, 2001)

52. Klemmer, S.R., M. Thomsen, E. Phelps-Goodman, R. Lee and J.A. Landay. Where Do Web Sites Come From? Capturing and Interacting with Design History. In the *Proceedings of the CHI 2002 Conference on Human Factors in Computing Systems:* pp. 1-8, Minneapolis, Minnesota, USA, ACM Press (April 20-25, 2002)

53. Lamming, M., P. Brown, K. Carter, M. Eldridge, M. Flynn, G. Louie, P. Robinson and A. Sellen, The Design of a Human Memory Prosthesis. *The Computer Journal **37**(3):* pp. 153-163 (1994)

54. Lamming, M. and M. Flynn. Forget-Me-Not: Intimate Computing in Support of Human Memory. In the *Proceedings of Friend 21: International Symposium on Next Generation Human Interface:* pp. 125-128, Meguro Gajoen, Japan (February 2-4, 1994)

55. Lamming, M.G. NoTime - A Tool for Notetakers. EuroPARC Technical Report (1991).

56. Landay, J.A. and R.C. Davis. Making Sharing Pervasive: Ubiquitous Computing

for Shared Note Taking. *IBM Systems Journal* ***38****(4):* pp. 531-550 (1999)

57. Licklider, J.C.R. Man Machine Symbiosis. *IRE Transactions on Human Factors in Electronics* ***HFE****(1):* pp. 4-11 (1960)

58. Lin, M., M.W. Newman, J.I. Hong and J.A. Landay. DENIM: finding a tighter fit between tools and practice for Web site design. In the *Proceedings of the 2000 Conference on Human Factors in Computing Systems:* pp. 510-517, The Hague, The Netherlands, ACM Press (April 1-6, 2000)

59. Lin, M., W.G. Lutters and T.S. Kim. Understanding the Micronote Lifecycle: Improving Mobile Support for Informal Note Taking. In the *Proceedings of the 2004 Conference on Human Factors in Computing Systems:* pp. 687-694, Vienna, Austria, ACM Press (April 24-29, 2004)

60. Liu, Q., Y. Rui, A. Gupta and J.J. Cadiz. Automating Camera Management for Lecture Room Environments. In the *Proceedings of the Conference on Human Factors in Computing Systems:* pp. 442-449, Seattle, Washington, USA, ACM Press (March 31-April 5, 2001)

61. Lockerd, A. and F.M. Mueller. LAFCam: Leveraging Affective Feedback Camcorder. In the *Extended Abstracts of the CHI 2002 Conference on Human Factors in Computing Systems:* pp. 574-575, Minneapolis, Minnesota, USA, ACM Press (April 20-25, 2002)

62. Long, A.C. Quill: A Gesture Design Tool for Pen-Based User Interfaces. Ph.D. Dissertation: 292 pages, University of California at Berkeley, Berkeley, California, USA (2001)

63. Macedo, A.A., K.N. Truong, J.A. Camacho-Guerrero and M.d.G. Pimentel. Automatically Sharing Web Experiences through a Hyperdocument Recommender System. In the *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia:* pp. 48-56, Nottingham, UK, ACM Press (August 26-30, 2003)

64. MacIntyre, B., E.D. Mynatt, S. Voida, K.M. Hansen, J. Tullio and G.M. Corso. Support for Multitasking and Background Awareness Using Interactive Peripheral Displays. In the *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology:* pp. 41-50, Orlando, Florida, USA, ACM Press (November 11-14, 2001)

65. Mankoff, J.C. An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based Input. Ph.D. Dissertation: 188 pages, Georgia Institute of Technology, Atlanta, Georgia, USA (2001)

66. McGee, D.R., P.R. Cohen and L. Wu. Something from Nothing: Augmenting a Paper-Based Work Practice via Multimodal Interaction. In the *Proceedings of DARE 2000 on Designing Augmented Reality Environments:* pp. 71-80, Elsinore, Denmark, ACM Press (April 12-14, 2000)

67. Melenhorst, A.S., Fisk, A.D., Mynatt, E.D. and Rogers, W.A. Potential intrusiveness of aware home technology: Perceptions of older adults. In the *Proceedings of Human Factors and Ergonomics Society 48th Annual Meeting:* New Orleans, Louisiana, USA, (September 20-24, 2004).

68. Minneman, S.L., S.R. Harrison, B. Janssen, G. Kurtenbach, T.P. Moran, I.E. Smith and W.v. Melle. A Confederation of Tools for Capturing and Accessing Collaborative Activity. In the *Proceedings of the Third ACM International Conference on Multimedia:* pp. 523-534, San Francisco, California, USA, ACM Press (November 5-9, 1995)

69. Minneman, S.L. and S.R. Harrison. Where Were We: Making and Using Near-Synchronous, Pre-Narrative Video. In the *Proceedings of the First ACM International Conference on Multimedia:* pp. 207-214, Anaheim, California, USA, ACM Press (August 1-6, 1993)

70. Moran, T.P., W.v. Melle and P. Chiu. Spatial Interpretation of Domain Objects Integrated into a Freeform Electronic. In the *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology:* pp. 175-184, San Francisco, California, USA, ACM Press (November 1-4, 1998)

71. Moran, T.P., L. Palen, S. Harrison, P. Chiu, D. Kimber, S. Minneman, W.v. Melle and P. Zellweger. "I'll get that off the audio": A Case Study of Salvaging Multimedia Meeting Records. In the *Proceedings of the SIGCHI conference on Human Factors in Computing Systems:* pp. 202-209, Atlanta, Georgia, USA, ACM Press (March 22-27, 1997)

72. Mukhopadhyay, S. and B. Smith. Passive capture and structuring of lectures. In the *Proceedings of the Seventh ACM International Conference on Multimedia:* pp. 477-487, Orlando, Florida, USA, ACM Press (October 30-November 5, 1999)

73. Myers, B.A. and M.B. Rosson. Survey on User Interface Programming. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 195-202, Monterey, California, USA, ACM Press (May 3-7, 1992)

74. Myers, B.A., H. Stiel and R. Gargiulo. Collaboration Using Multiple PDAs Connected to a PC. In the *Proceedings of the ACM Conference on Computer Supported Cooperative Work:* pp. 285-294, Seattle, Washington, USA, ACM Press (November 14-18, 1998)

75. Mynatt, E.D., T. Igarashi, W.K. Edwards and A. LaMarca. Flatland: New Dimensions in Office Whiteboards. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 346-353, ACM Press: Pittsburgh, Pennsylvania, USA, ACM Press (May 15-20, 1999)

76. Newman, W. and P. Wellner. A Desk Supporting Computer-Based Interaction with Paper Documents. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 587-592, Monterey, California, USA, ACM Press (May 3-7, 1992)

77. Newman, W.M., M.A. Eldridge and M.G. Lamming, PEPYS: Generating Autobiographies by Automatic Tracking. In the *Proceedings of the 2nd European Conference on Computer-Supported Cooperative Work:* pp. 175-188, Amsterdam, The Netherlands, Kluwer (September 24-27, 1991)

78. Patel, S.N. and G.D. Abowd. The ContextCam: Automated Point of Capture Video Annotation. In the *Proceedings of the 6th International Conference on Ubiquitous Computing:* pp. 301-318, Nottingham, UK, Springer-Verlag (September 7-10, 2004)

79. Pedersen, E.R., K. McCall, T.P. Moran and F.G. Halasz. Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. In the *Proceedings of the Conference on Human Factors in Computing Systems:* pp. 391-398, Amsterdam, The Netherlands, ACM Press (April 24-29, 1993)

80. Pimentel, M.d.G.C., G.D. Abowd and Y. Ishiguro. Linking by Interacting: A Paradigm for Authoring Hypertext. In the *Proceedings of the Eleventh ACM Conference on Hypertext and Hypermedia:* pp. 39-48, San Antonio, Texas, USA, ACM Press (May 30-June 3, 2000)

81. Pimentel, M.d.G.C., Y. Ishiguro, B. Kerimbaev, G.D. Abowd and M. Guzdial. Supporting Educational Activities through Dynamic Web Interfaces. *Interacting with Computers, **13**(3):* pp. 353-374 (2001)

82. Rekimoto, J. Time-Machine Computing: A Time-Centric Approach for the Information Environment. In the *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology:* pp. 45-54, Asheville, North Carolina, USA, ACM Press (November 7-10, 1999)

83. Rhodes, B.J. and T. Starner. Remembrance Agent: A Continuously Running Automated Information Retrieval System. In the *Proceedings of the 1st International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology:* pp. 487-495, London, UK (April 22-24, 1996).

84. Richter, H., P. Schuchhard and G.D. Abowd. Automated capture and retrieval of architectural rationale. Technical Report GIT-GVU-98-37: Georgia Institute of Technology, Atlanta, Georgia, USA (1999).

85. Richter, H.A. Designing and Evaluating Meeting Capture and Access Services. Ph.D. Dissertation: 194 pages, Georgia Institute of Technology, Atlanta, Georgia, USA (2005)

86. Richter, H.A., G.D. Abowd, W. Geyer, L. Fuchs, S. Daijavad and S.E. Poltrock. Integrating Meeting Capture within a Collaborative Team Environment. In the *Proceedings of the 3rd International Conference on Ubiquitous Computing:* pp. 123-138, Atlanta, Georgia, USA, Springer-Verlag (September 31-October 2, 2001)

87. Rist, T., J.-C. Martin, F. Néel and J. Vapillon. On the Design of Intelligent Memory Functions for Virtual Meeting Places: Examining Potential Benefits and Requirements. *Journal Le Travail Humain,* ***63****(3):* pp. 203-225 (2000)

88. Sarvas, R., E. Herrarte, A. Wilhelm and M. Davis. Metadata Creation System for Mobile Images. In the *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services:* pp. 36-48, Boston, Massachusetts, USA, ACM Press (June 6-9, 2004)

89. Satyanarayanan, M. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications,* ***8****(4):* pp. 10-17 (2001)

90. Schilit, B.N., N. Adams, R. Gold, M.M. Tso and R. Want. The PARCTAB Mobile Computing System. In the *Proceedings of Workshop on Workstation Operating Systems:* pp. 34-39, Napa, California, USA, IEEE Computer Society Press (October 14-15, 1993)

91. Stafford-Fraser, Q. and P. Robinson. BrightBoard: A Video-Augmented Environment. In the *Proceedings of Conference on Human Factors in Computing Systems:* pp. 134-141, Vancouver, British Columbia, Canada, ACM Press (April 13-18, 1996)

92. Stifelman, L.J. The Audio Notebook. Ph.D. Dissertation: 150 pages, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA (1997)

93. Stifelman, L.J., B. Arons and C. Schmandt. The Audio Notebook: Paper and Pen Interaction with Structured Speech. In the *Proceedings of the Conference on Human Factors in Computing Systems:* pp. 182-189, Seattle, Washington, USA, ACM Press (March 31-April 5, 2001)

94. Streitz, N.A., J. Geißler, J.M. Haake and J. Hol. DOLPHIN: Integrated Meeting

Support Across Local and Remote Desktop Environments and LiveBoards. In the *Proceedings of the Conference on Computer Supported Cooperative Work:* pp. 345-358, Chapel Hill, North Carolina, USA, ACM Press (October 22-26, 1994)

95. Sumi, Y., T. Etani, S. Fels, N. Simonet and K. Mase. C-MAP: Building a context-aware mobile assistant for exhibition tours. *Community Computing and Support Systems. LCNS 1519:* pp. 137-154, Springer-Verlag (1998).

96. Sumi, Y., R. Sakamoto, K. Nakao and K. Mase. ComicDiary: Representing Individual Experiences in a Comics Style. In the *Proceedings of the 4th International Conference on Ubiquitous Computing:* pp. 16-32, Göteborg, Sweden, Springer-Verlag (September 29-October 1, 2002)

97. Tran, Q.T., G. Calcaterra and E.D. Mynatt. Cook's Collage: Déjà vu Display for a Home Kitchen. In the *Proceedings of HOIT 2005 (Home Oriented Informatics and Telematics):* York, U.K., (April 13-15, 2005)

98. Tran, Q.T. and E.D. Mynatt. What Was I Cooking? Towards Déjà Vu Displays of Everyday Memory. Technical Report GIT-GVU-03-33: Georgia Institute of Technology, Atlanta, Georgia, USA (2003).

99. Truong, K.N. and G.D. Abowd. INCA: A Software Infrastructure to Facilitate the Construction and Evolution of Ubiquitous Capture & Access Applications. In the *Proceedings of the 2nd International Conference on Pervasive Computing:* pp. 140-157, Vienna, Austria, Springer-Verlag (April 21-23, 2004)

100. Truong, K.N., G.D. Abowd and J.A. Brotherton. Personalizing the Capture of Public Experiences. In the *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology:* pp. 121-130, Asheville, North Carolina, USA, ACM Press (November 7-10, 1999)

101. Truong, K.N., G.D. Abowd and J.A. Brotherton. Who, What, When, Where, How: Design Issues of Capture & Access Applications. In the *Proceedings of the 3rd International Conference on Ubiquitous Computing:* pp. 209-224, Atlanta, Georgia, USA, Springer-Verlag (September 31-October 2, 2001)

102. Truong, K.N., E.M. Huang, and G.D. Abowd. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. In the *Proceedings of the 6th International Conference on Ubiquitous Computing:* pp. 143-160, Nottingham, UK, Springer-Verlag (September 7-10, 2004)

103. Truong, K.N., S.N. Patel, J.W. Summet and G.D. Abowd. Preventing Camera Recording by Designing a Capture-Resistant Environment. In the *Proceedings of the 7th International Conference on Ubiquitous Computing:* Tokyo, Japan,

Springer-Verlag (September 11-14, 2005)

104. Uchihashi, S., J. Foote, A. Girgensohn and J. Boreczky, Video Manga: Generating Semantically Meaningful Video Summaries. In the *Proceedings of the Seventh ACM International Conference on Multimedia:* pp. 383-392, Orlando, Florida, USA, ACM Press (October 30-November 5, 1999)

105. Want, R., A. Hopper, V. Falcão and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems, **10**(1):* pp. 91-102 (1992)

106. Want, R., T. Pering, G. Danneels, M. Kumar, M. Sundar and J. Light. The Personal Server: Changing the Way We Think about Ubiquitous Computing. In the *Proceedings of the 4th International Conference on Ubiquitous Computing:* pp. 194-209, Göteborg, Sweden, Springer-Verlag (September 29-October 1, 2002)

107. Weiser, M. The Computer for the 21st Century. *Scientific American, **265**(30):* pp. 94-104 (1991)

108. Weiser, M. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM, **36**(7):* pp. 75-84 (1993).

109. White, D.R., J.A. Camacho-Guerrero, K.N. Truong, G.D. Abowd, M.J. Morrier, P.C. Vekaria and D. Gromala. Mobile Capture and Access for Assessing Language and Social Development in Children with Autism. In the *Extended Abstracts of the 5th International Conference on Ubiquitous Computing:* pp. 137-140, Seattle, Washington, USA (October 12-15, 2003)

110. Whittaker, S., P. Hyland and M. Wiley. FILOCHAT: Handwritten Notes Provide Access to Recorded Conversations. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 271-276, Boston, Massachusetts, USA, ACM Press (April 24-28, 1994)

111. Wilcox, L.D., B.N. Schilit and N. Sawhney. Dynomite: A Dynamically organized Ink and Audio Notebook. In the *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems:* pp. 186-193, Atlanta, Georgia, USA, ACM Press (March 22-27, 1997)